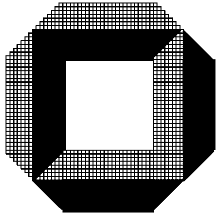


Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation

Erweiterbare Übersetzer

Diplomarbeit

Matthias Zenger

August 1998

Betreuer: Prof. Martin Odersky
University of South Australia

Prof. Walter Tichy
Universität Karlsruhe

Hiermit erkläre ich, die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Matthias Zenger Karlsruhe, 08. August 1998

Kurzfassung

Übersetzer sind im allgemeinen komplexe Systeme, die sehr schwer zu warten und zu erweitern sind. Vor allem bei Übersetzern für eine Familie von Programmiersprachen sind Modularisierung und Wiederverwendung von Datenstrukturen und Übersetzerkomponenten besonders wichtig. Der traditionelle Übersetzerbau kennt jedoch keine Konzepte für die Erweiterbarkeit oder Wiederverwendbarkeit von Übersetzern auf der Implementationsebene. Bestenfalls spezielle Werkzeuge unterstützen die Wiederverwendung von Übersetzerkomponenten. In dieser Arbeit werden Konzepte untersucht, mittels der erweiterbare Übersetzer unabhängig von Werkzeugen implementiert werden können.

Üblicherweise wird der Übersetzungsvorgang in eine sequentielle Folge von Phasen gegliedert. Jede Phase stellt eine Operation auf einer rekursiven Datenstruktur, der Programmrepräsentation dar. Erweiterbare Übersetzer erfordern eine gleichzeitige und unabhängige Erweiterbarkeit von Programmrepräsentation und Operationen. Bisherige Entwurfsstrategien unterstützen eine Erweiterbarkeit in beide Dimensionen allerdings nur unzureichend. Das Problem läßt sich elegant durch die Verwendung *erweiterbarer algebraischer Datentypen* lösen. Diese erlauben auf einfache Art und Weise

- Erweiterungen von Datentypen,
- Veränderungen bestehender Operationen und
- die Implementierung neuer Operationen auf einem Datentyp.

Basierend auf diesen Typen wird eine Software-Architektur für frei erweiterbare Übersetzer vorgeschlagen, in der beliebige Erweiterungen bzw. Veränderung vorgenommen werden können, ohne daß Modifikationen an bestehenden Quelltexten nötig werden. Ein erweiterter Übersetzer wird vollständig auf den Komponenten seines Vorgängers aufgebaut. Er erweitert diesen durch neue Komponenten oder ersetzt Komponenten durch neue, modifizierte Versionen. Damit ist eine Grundlage geschaffen, auf der Übersetzer für aufeinander aufbauende Programmiersprachen wie beispielsweise Pizza und Java inkrementell entwickelt und gemeinsam unterhalten werden können.

Danksagung

Zunächst möchte ich mich ganz herzlich bei meinem Betreuer Prof. Martin Odersky bedanken, der mich während meiner gesamten Arbeitszeit persönlich unterstützt hat, stets offen für Fragen und Diskussionen war und von dem ich unzählige Hinweise und Verbesserungsvorschläge erhalten habe. Mein Dank gilt auch der School of Computer and Information Science der University of South Australia, bei der ich als Gast den praktischen Teil meiner Arbeit realisieren konnte. Während meines Aufenthalts in Australien haben mich, vor allem was organisatorische und technische Fragen betriff, Dr. John Maraist, Alex Cowie, Randall Fletcher, Enno Runne und Oliver Reiff unterstützt. Für die Korrektur der Ausarbeitung und für zahlreiche Anregungen bezüglich der Darstellung der Thematik, bedanke ich mich bei Stephan Michael Bischoff, Matthias John, Raphael Straub und Dr. Michael Philippsen. Schließlich gilt mein Dank Prof. Walter Tichy, der sich bereit erklärt hat, neben Prof. Martin Odersky meine Diplomarbeit zu betreuen.

Matthias Zenger Karlsruhe, im August 1998

Inhaltsverzeichnis

1 Erweiterbarkeit im Übersetzerbau **1**
- 1.1 Wiederverwendung von Übersetzern . . . 1
- 1.2 Übersetzer-Architekturen . . . 2
- 1.3 Anforderungen an frei erweiterbare Übersetzer . . . 4
- 1.4 Einordnung der Arbeit . . . 5
- 1.5 Überblick . . . 5

2 Erweiterbare Interpreter **7**
- 2.1 Objektorientierte Interpreter . . . 8
- 2.2 Visitors . . . 11
  - 2.2.1 Klassische Visitors . . . 11
  - 2.2.2 Erweiterung eines Visitors . . . 12
  - 2.2.3 Alternative Realisierungen erweiterbarer Visitors . . . 14
- 2.3 Typschalter . . . 16
- 2.4 Funktionale Interpreter . . . 19
  - 2.4.1 Interpreter in funktionalen Sprachen . . . 19
  - 2.4.2 Erweiterung funktionaler Interpreter . . . 20
- 2.5 Zusammenfassung . . . 21
- 2.6 Benchmarks . . . 22

3 Erweiterbare Algebraische Typen **27**
- 3.1 Algebraische Datentypen . . . 27
- 3.2 Erweiterbare Union-Types . . . 28
  - 3.2.1 Geschlossene Summentypen . . . 28
  - 3.2.2 Bisherige Ansätze . . . 29
- 3.3 Erweiterbare Algebraische Typen . . . 30
  - 3.3.1 Offene Summentypen . . . 30
  - 3.3.2 Konsequenzen . . . 32
- 3.4 Erweiterbare Algebraische Typen für Java . . . 33
  - 3.4.1 Syntax und Semantik . . . 34
  - 3.4.2 Typsystem . . . 41
- 3.5 Übersetzung erweiterbarer algebraischer Typen . . . 44
  - 3.5.1 Deklarationen . . . 44
  - 3.5.2 Typen . . . 45

3.5.3 Typoperatoren 47
3.5.4 Statische Qualifikationen 50
3.6 Übersetzung von Pattern Matching 51
3.6.1 Übersetzung von `switch`-Blöcken 52
3.6.2 Generierung von Java-Code 62
3.6.3 Vollständigkeitsprüfung für Fallunterscheidungen 64

4 Erweiterbare Übersetzer **71**
4.1 Grundkonzepte der Software-Architektur 71
4.2 Architekturmuster *Context-Component* 73
4.2.1 Motivation 73
4.2.2 Idee 74
4.2.3 Struktur 74
4.2.4 Konsequenzen 76
4.2.5 Implementierung 78
4.2.6 Verwandte Muster 80
4.3 Architekturstil *Batch-sequentielles Repository* 80
4.4 Erweiterbare Übersetzer-Architektur 81
4.4.1 Komponenten 82
4.4.2 Dekomposition von Phasen 82
4.4.3 Strukturelle Dekomposition von Phasen 85
4.4.4 Funktionale Dekomposition von Phasen 85
4.4.5 Repräsentation von Daten 88
4.4.6 Erweiterung eines Übersetzers 90
4.5 Ein erweiterbarer Java-Übersetzer 91
4.5.1 Architektur des Java-Übersetzers 91
4.5.2 Erweiterung des Übersetzers 98

5 Zusammenfassung **101**
5.1 Beiträge der Arbeit 101
5.2 Praktische Arbeiten 102
5.3 Ausblick 103

Literaturverzeichnis **105**

Kapitel 1

Erweiterbarkeit im Übersetzerbau

1.1 Wiederverwendung von Übersetzern

Bei der Entwicklung eines Übersetzers geht man traditionellerweise davon aus, daß der Übersetzer eine bestimmt, feste Quellsprache zu übersetzen hat. Erweiterbarkeit und Wiederverwendbarkeit spielen deswegen für die meisten Komponenten eines Übersetzers keine Rolle. Bestenfalls das Backend wird so gestaltet, daß es gegen eine Version für eine andere Zielsprache ausgetauscht werden kann. In der Praxis werden Übersetzer aber oftmals für Programmiersprachen geschrieben, deren endgültige Definition noch nicht abgeschlossen ist. Abhängig von den Erfahrungen, die mit einer Sprache in der Praxis gesammelt werden und den Bedürfnissen und Anforderungen ihrer Benutzer, werden neue Sprachkonstrukte hinzugefügt, bestehende verändert oder wieder aus der Programmiersprache herausgenommen. Bei diesem Prozeß entsteht eine *Familie von verwandten Programmiersprachen.*

Parallel zu der Entwicklung der Sprachen müssen geeignete Übersetzer geschrieben werden. Im traditionellen Übersetzerbau [ASU92, WG84] gibt es jedoch keine Konzepte für die *Erweiterbarkeit* oder *Wiederverwendbarkeit* von Übersetzern. Selbst wenn Quellcode oder Spezifikationen eines existierenden Übersetzers vorliegen, bedeutet das Schreiben eines neuen Übersetzers für eine erweiterte Sprache oftmals einen äußerst großen Aufwand. Neben dem Problem der Erweiterbarkeit stellt sich dann auch die Frage des *Unterhalts* von Übersetzern für verwandte Programmiersprachen. Änderungen im Übersetzer für die Basissprache schlagen sich nicht automatisch in den Übersetzern für die erweiterten Sprachen nieder und müssen dort per Hand vorgenommen werden. Inkonsistenzen zwischen den einzelnen Übersetzern ergeben sich auf diese Weise nahezu zwangsläufig.

Ein Beispiel für diese Problematik stellen die Java-Übersetzer dar, die am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelt wurden. Der Übersetzer EspressoGrinder [OP95, Zen96], der ursprünglich Java 1.0 beta Code übersetzte, wurde zunächst um Funktionen höherer Ordnung ergänzt. Dieser Übersetzer war dann Ausgangspunkt für eine ganze Reihe weiterer Übersetzer für verschiedene

erweiterte Java-Dialekte. Neben dem Übersetzer für die Sprache E [EC96] von Electric Communities basiert auch der Übersetzer von JavaParty [Zen97, PZ97], einer transparent verteilten Java Programmierumgebung, auf EspressoGrinder. Pizza [OW97], eine Obermenge von Java 1.1, bietet zusätzlich parametrische Polymorphie und algebraische Datentypen. Der Übersetzer für Pizza ist zwar selbst in Pizza geschrieben, wurde aber ausgehend vom EspressoGrinder-Code implementiert. Abbildung 1.1 gibt einen Überblick über die Evolution der Sprachen bzw. Übersetzer.

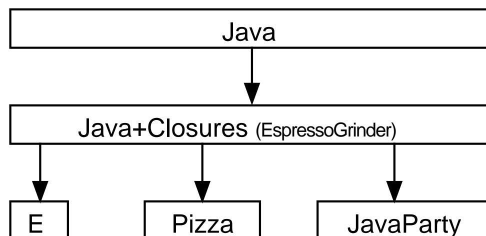


Abbildung 1.1: Familie von Java-Dialekten

Die einzelnen Übersetzer wurden ursprünglich alle auf dem Quelltext von EspressoGrinder aufgebaut, wobei die Behandlung der Spracherweiterungen jeweils direkt in den Quelltext eingefügt wurde. Damit haben sich die verschiedenen Übersetzer, obwohl sie alle auf dem gleichen Basissystem aufbauen, zu eigenständigen, unabhängigen Systemen entwickelt, die alle getrennt gewartet werden müssen. Fehler die im Basisübersetzer gefunden werden, müssen beispielsweise in allen davon abgeleiteten Versionen einzeln beseitigt werden.

Die Entwicklung einer Familie von Programmiersprachen ist, wie Abbildung 1.1 verdeutlicht, ein inkrementeller Prozeß. Es liegt nahe, auch auf der Übersetzerseite inkrementell eine zur Sprachhierarchie analoge Übersetzerhierarchie aufzubauen, in der ein Übersetzer die Komponenten seines Vorgängers wiederverwendet bzw. erweitert, ohne den Code des Vorgängers selbst zu verändern. Die Erweiterungen eines Übersetzers teilen sich damit die Komponenten ihres gemeinsamen Vorgängers. Hiermit ist eine Grundlage geschaffen, auf der die verschiedenen Übersetzer gemeinsam unterhalten werden können.

1.2 Übersetzer-Architekturen

In [Bos96a, Bos96b] wird neben den Problemen der Erweiterbarkeit, Wiederverwendbarkeit und Unterhaltbarkeit im traditionellen Übersetzerbau auch die Komplexität von Übersetzern als ein zentrales Problem beschrieben. Üblicherweise versucht man dieses Problem durch eine Dekomposition des Übersetzungsvorgangs in eine Sequenz von aufeinanderfolgenden *Phasen* zu lösen, die über verschiedene Zwischenrepräsentationen des zu übersetzenden Programms gekoppelt werden. Abbildung 1.2 zeigt die traditionelle Struktur eines Übersetzers.

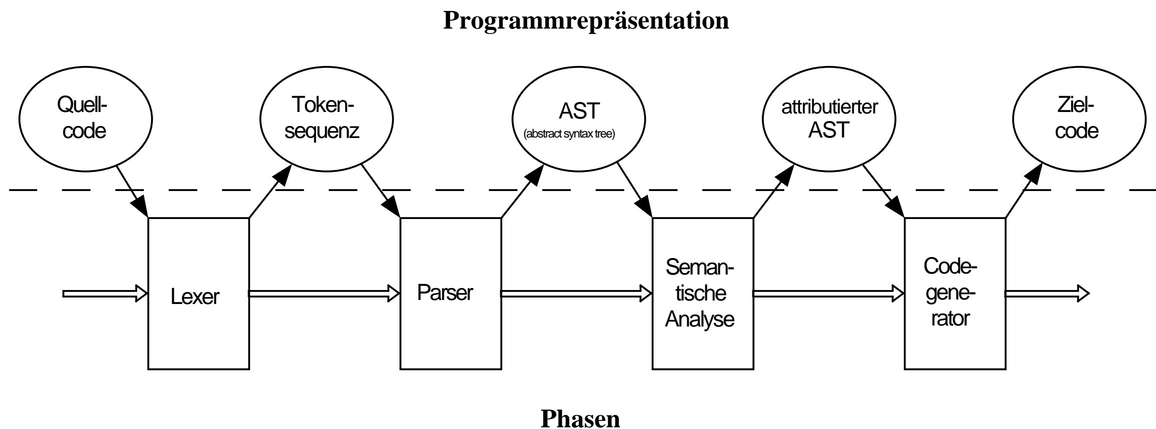


Abbildung 1.2: Traditionelle Struktur eines Übersetzers

In dieser *batch-sequentiellen* Übersetzerarchitektur transformiert jede Phase ein Programm von einer Repräsentation in eine andere. Die einzelnen Zwischensprachen sind heutzutage allerdings nur konzeptioneller Natur. Moderne Übersetzer besitzen eine zentrale interne Programmrepräsentation in Form eines abstrakten Syntaxbaums, dem sogenannten Strukturbaum. Dieser wird zu Beginn des Übersetzungsvorgangs erzeugt und in den darauffolgenden Phasen kontinuierlich verändert. Abbildung 1.3 zeigt das Modell eines solchen Übersetzers. Aus der Sicht der Software-Architektur handelt es sich bei diesem Modell um eine Variante eines *Repositories* [SG96]. Im Mittelpunkt steht die zentrale Programmrepräsentation, die von den sequentiell ausgeführten Phasen jeweils interpretiert und modifiziert wird. Neben der Programmrepräsentation gibt es noch eine Reihe von weiteren globalen Datenstrukturen auf die von den einzelnen Phasen aus zugegriffen wird.

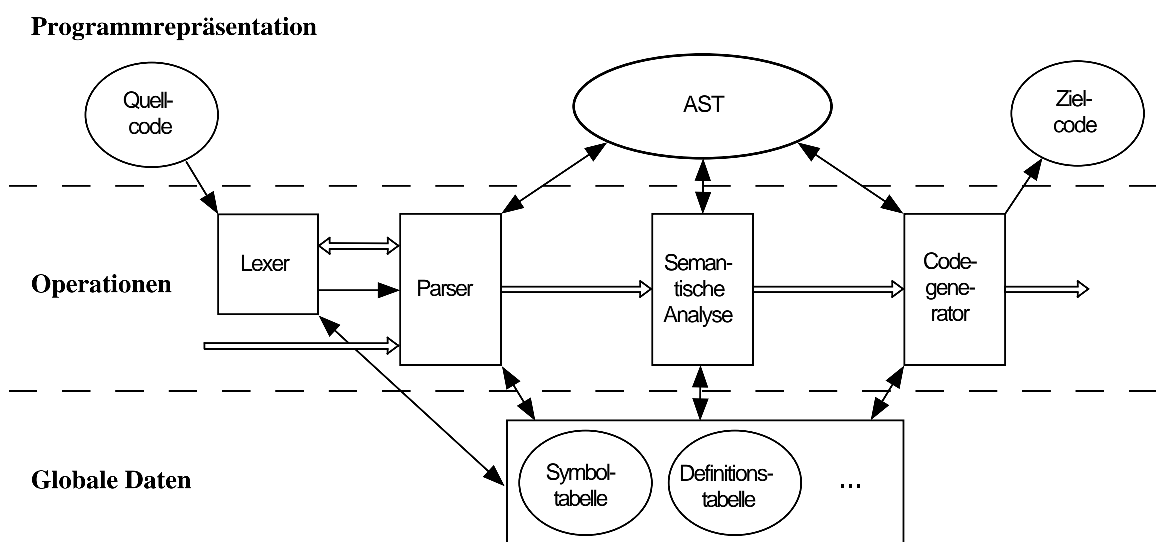


Abbildung 1.3: Struktur eines modernen Übersetzers

Unabhängig davon, ob ein Übersetzer strikt batch-sequentiell oder als Repository aufgebaut wird, ist die Organisationsform des Systems funktionaler Natur: die Programmrepräsentation kann als *Datum* gesehen werden, wohingegen die einzelnen Phasen *Funktionen* darstellen, die auf diesen Daten operieren. Für erweiterbare Übersetzer ist es erforderlich, daß man zugleich Programmrepräsentation sowie Operationen auf der Programmrepräsentation erweitern kann. Erweiterungen finden bei einem Übersetzer nämlich gewöhnlicherweise zweistufig statt: zum einen wird die Syntax erweitert, um neue syntaktische Konstrukte hinzuzufügen, zum anderen werden neue Operationen für den Strukturbaum benötigt, welche beispielsweise als Phasen die Semantik der Spracherweiterungen behandeln.

1.3 Anforderungen an frei erweiterbare Übersetzer

Übersetzer sind im allgemeinen große und komplexe Software-Systeme. Grundlage für die Erweiterbarkeit eines Übersetzers ist eine geeignete funktionale und strukturelle Gliederung in Teilkomponenten. Eine solche Modularisierung reduziert zudem die Komplexität im System und fördert damit das Verständnis für dessen Aufbau und Funktionsweise. Ein System das nicht verstanden wird, kann auch nicht wiederverwendet oder erweitert werden.

Erweiterbarkeit ist in einem Übersetzer in mehrerlei Hinsicht von Bedeutung. Unter einem *frei erweiterbaren Übersetzer* wird ein Übersetzer verstanden, bei dem es zugleich möglich ist

- Datentypen zu erweitern (z.B. zur Repräsentation des Strukturbaums),
- bestehende Komponenten zu verändern (z.B. existierende Operationen bezüglich erweiterter Datentypen anzupassen) und
- neue Komponenten hinzuzufügen.

Um einen Übersetzer flexibel wiederverwenden zu können, müssen die Erweiterungen so durchgeführt werden können, daß

- neben den erweiterten Komponenten auch noch auf die alten Komponenten zugegriffen werden kann und
- am Code des ursprünglichen Übersetzers keine Modifikationen vorgenommen werden müssen.

1.4 Einordnung der Arbeit

Diese Arbeit beschäftigt sich mit den Grundlagen erweiterbarer Übersetzer auf der Implementationsebene. Die allgemeine Erweiterbarkeit von Übersetzern wurde bisher, wenn überhaupt, nur in Zusammenhang mit speziellen Werkzeugen diskutiert, die es ermöglichen sollen, Komponenten oder Spezifikationen von Übersetzern wiederzuverwenden. Übersetzerbau-Werkzeuge, die die Wiederverwendung von Übersetzer-Komponenten unterstützen, sind nicht Gegenstand der vorliegenden Untersuchungen.

Erweiterbare Übersetzer spielen auch im Zusammenhang mit *erweiterbaren Programmiersprachen* eine Rolle. Hier wird allerdings nur eine begrenzte und durch die Sprache genau definierte Erweiterbarkeit des Übersetzers benötigt [Bos97]. Spracherweiterungen lassen sich mittels einer Metasprache spezifizieren, welche vom Übersetzer als eine Art Plug-In verwendet wird [TC97, IR97]. Die Erweiterbarkeit ist Teil der Sprache und wird bereits beim Schreiben des Basisübersetzers mit berücksichtigt. Übersetzer für erweiterbare Sprachen besitzen meist eine einzige erweiterbare Komponente: einen Präprozessor. Die Metasprache spezifiziert Erweiterungen einer Programmiersprache durch eine Programmtransformation in die Basissprache. Erweiterbarkeit wird in dieser Arbeit wesentlich allgemeiner verstanden. Sie darf sich nicht auf Mechanismen beschränken, die beim Entwurf eines Übersetzers explizit im Rahmen der Quellsprache vorgesehen sind. Übersetzer für erweiterbare Programmiersprachen sind deswegen nicht notwendigerweise auch erweiterbare Übersetzer in dem hier verwendeten allgemeinen Sinn.

1.5 Überblick

In Kapitel 2 werden zunächst verschiedene Ansätze diskutiert, wie man Strukturbäume und Operationen darauf erweiterbar implementieren kann. Traditionelle Entwurfsstrategien unterstützen eine Erweiterbarkeit in beide Dimensionen nur unzureichend. *Erweiterbare algebraische Typen* sind dagegen nicht mit dieser Problematik behaftet. Kapitel 3 beschreibt die theoretischen Aspekte dieser neuen Datentypen und erläutert, wie man erweiterbare algebraische Typen in Java integrieren kann. Es wird zudem diskutiert, wie erweiterbare algebraische Typen in reguläres Java übersetzt werden können. In Kapitel 4 wird zunächst ein Architekturmuster vorgestellt, mit dem auf flexible Art und Weise erweiterbare Komponentensysteme aufgebaut werden können. Es folgt eine kurze Beschreibung des gängigen Architekturstils für Übersetzter. Schließlich wird diskutiert, wie frei erweiterbare Übersetzer diesem Architekturstil folgend, mit Hilfe des allgemeinen Architekturmusters aufgebaut werden können. Erweiterbare algebraische Typen dienen dabei zur übersetzerinternen Repräsentation von Daten. Abschnitt 4.5 beschreibt abschließend die Software-Architektur eines frei erweiterbaren Java-Übersetzers, der im Rahmen dieser Arbeit implementiert wurde. Der Übersetzer wurde so erweitert, daß er erweiterbare algebraische Typen in Java unterstützt.

Kapitel 2

Erweiterbare Interpreter

Der Strukturbaum innerhalb eines Übersetzers wird durch eine rekursive Datenstruktur implementiert. Die einzelnen Phasen eines Übersetzers stellen, zumindest strukturell, rekursive Operationen auf dieser Datenstruktur dar. Um erweiterbare Übersetzer zu bauen, muß es zugleich möglich sein, Datentypen und Operationen zu erweitern, sowie neue Typen und Operationen zu definieren, ohne Modifikationen am Quelltext vornehmen zu müssen. Wie man sehen wird, lösen bestehende Entwurfsstrategien diese Art von Problem nur sehr unzureichend [KFF98]. Erweiterbare algebraische Datentypen erlauben es dagegen recht elegant und effizient, Implementierungen für solche Aufgabenstellungen zu formulieren.

Übersetzerphasen sind nichts anderes als Interpreter des Strukturbaums. Ein geeignetes Entwurfsmuster für einen erweiterbaren Interpreter ist also Grundvoraussetzung um erweiterbare Übersetzer bauen zu können. In diesem Kapitel werden verschiedene Entwurfsstrategien zur Implementierung von Interpretern für eine einfache Sprache vorgestellt. Die Sprache wird im nachhinein um λ-Abstraktionen erweitert. Es wird jeweils untersucht, inwieweit es möglich ist, bestehende Interpreter entsprechend zu erweitern bzw. neue Interpreter für die erweiterte Sprache zu schreiben. Es wird darauf Wert gelegt, daß sämtliche Änderungen und Erweiterungen möglichst ohne Modifikationen am Quelltext vorgenommen werden können. In Abbildung 2.1 wird die abstrakte Syntax und die denotationelle Semantik der vollständigen Beispielsprache definiert.

In der Definition der Semantik repräsentiert ρ eine Umgebung, die Variablenbindungen enthält. Um eine neue Bindung einer Variable x an einen Wert v hinzuzufügen, würde man $[x \mapsto v]\,\rho$ schreiben. Den an eine Variable x in ρ gebundenen Wert erhält man über den Ausdruck $\rho(x)$. Die Bedeutungsfunktion $[\![\cdot]\!]$ definiert für einen Ausdruck eine Abbildung von Umgebungen auf Werte.

Für die einzelnen Entwurfsstrategien wird jeweils eine vollständige Implementierung eines Interpreters angegeben, der Ausdrücke der Beispielsprache gemäß der in Abbildung 2.1 definierten denotationellen Semantik auswertet. Die Programme werden in der

Abstrakte Syntax

Zahlen	n		
Variablen	x		
Ausdrücke	E	$=$	$N \mid A$
Arithmetische Ausdrücke	N	$=$	$E + E \mid E * E \mid x \mid n$
Abstraktion und Applikation	A	$=$	$\lambda x.E \mid E\,E$

Semantik

$$
\begin{array}{lcl}
[\![n]\!]\,\rho & = & n \\
[\![x]\!]\,\rho & = & \rho(x) \\
[\![e_1 + e_2]\!]\,\rho & = & \begin{cases} [\![e_1]\!]\,\rho \;+\; [\![e_2]\!]\,\rho & \text{falls } [\![e_1]\!]\,\rho \in \mathbb{Z} \text{ und } [\![e_2]\!]\,\rho \in \mathbb{Z} \\ \bot & \text{sonst} \end{cases} \\
[\![e_1 * e_2]\!]\,\rho & = & \begin{cases} [\![e_1]\!]\,\rho \;*\; [\![e_2]\!]\,\rho & \text{falls } [\![e_1]\!]\,\rho \in \mathbb{Z} \text{ und } [\![e_2]\!]\,\rho \in \mathbb{Z} \\ \bot & \text{sonst} \end{cases} \\
[\![\lambda x.e]\!]\,\rho & = & \lambda x.e \\
[\![e_1\,e_2]\!]\,\rho & = & \begin{cases} [\![e]\!]\,([x \mapsto [\![e_2]\!]\,\rho]\,\rho) & \text{falls } [\![e_1]\!]\,\rho = \lambda x.e \\ \bot & \text{sonst} \end{cases}
\end{array}
$$

Abbildung 2.1: Abstrakte Syntax und Semantik der Beispielsprache

Programmiersprache Java [GJS96] formuliert, sollten jedoch leicht auf eine andere objektorientierte Sprache übertragbar sein.

2.1 Objektorientierte Interpreter

Im objektorientierten Programmieren werden Datendefinitionen und Operationen auf den Daten parallel entwickelt. Jedes Konstrukt einer Sprache wird durch eine eigene Klasse repräsentiert. Abstrakte Oberklassen fassen mehrere Sprachkonstrukte zusammen, indem sie gemeinsame Daten definieren und die Signaturen für die anwendbaren Operationen spezifizieren. In konkreten Klassen für die einzelnen Sprachkonstrukte werden konstruktspezifische Daten gekapselt und die tatsächlichen Operationen bezüglich des jeweiligen Konstrukts implementiert. Diese Vorgehensweise entspricht genau dem Entwurfsmuster *Interpreter* [G⁺95]. Programm 2.1 zeigt eine mögliche Implementierung eines Interpreters für die Beispielsprache ohne λ-Abstraktionen.

Die abstrakte Oberklasse `Tree` aller Sprachkonstrukte definiert zwei Operationen `toInt` und `eval` und implementiert jeweils ein Standardverhalten für beide Methoden. Jedes Sprachkonstrukt wird durch eine konkrete Unterklasse repräsentiert. Diese Klassen definieren als Instanzvariablen jeweils die rechten Seiten der zugehörigen Produktion aus

```
abstract class Tree {
   int toInt() {
      throw new Error("number expected");
   }
   Tree eval(Env env) { return this; }
}
class Number extends Tree {
   int value;
   Number(int v) { value = v; }
   int toInt() { return value; }
}
class Variable extends Tree {
   String name;
   Variable(String n) { name = n; }
   Tree eval(Env env) {
      return env.lookup(name);
   }
}
class Plus extends Tree {
   Tree    left, right;
   Plus(Tree l, Tree r) { left = l; right = r; }
   Tree eval(Env env) {
      return new Number(left.eval(env).toInt() +
                        right.eval(env).toInt());
   }
}
class Times extends Tree {
   Tree    left, right;
   Times(Tree l, Tree r) { left = l; right = r; }
   Tree eval(Env env) {
      return new Number(left.eval(env).toInt() *
                        right.eval(env).toInt());
   }
}
```

```
class Env {
   String name;
   Tree   value;
   Env    next;
   Env(String name, Tree value, Env next) {
      this.name = name;
      this.value = value;
      this.next = next;
   }
   Tree lookup(String ident) {
      if (ident.equals(name))
         return value;
      else
         return next.lookup(ident);
   }
   static class Empty extends Env {
      Empty() { super(null, null, null); }
      Tree lookup(String ident) {
         throw new Error(ident + " unbound");
      }
   }
}
```

Programm 2.1: Objektorientierte Implementierung eines Interpreters

der Grammatik der abstrakten Syntax. Die Typen der Unterklassen werden als *Varianten*[1] von `Tree` bezeichnet. Die Umgebung ρ wird im Programm durch die Klasse `Env` dargestellt.

Dieser objektorientierte Ansatz erlaubt es relativ einfach, für eine vorgegebene Sprache eine entsprechende Klassenhierarchie aufzubauen. Weiterhin ist es sehr leicht möglich, neue Sprachkonstrukte nachträglich einzuführen und bestehende Operationen durch Unterklassenbildung zu modifizieren. Deswegen ist Programm 2.1 relativ einfach um λ-Abstraktionen erweiterbar. In Programm 2.2 ist der hierfür nötige Quelltext angegeben.

Das Entwurfsmuster Interpreter ist allerdings ungeeignet wenn es darum geht, neue Operationen nachträglich hinzuzufügen. Die einzige Möglichkeit dies ohne Veränderungen im Quelltext zu tun, besteht darin, ausnahmslos für alle konkreten Klassen Unterklassen anzulegen. In diesen Unterklassen wird dann jeweils die neue Operation implementiert. Diese Vorgehensweise hat zweifelsohne einige Schwächen:

1. Es ist äußerst aufwendig und ineffizient neue Operationen hinzuzufügen. Jedesmal entsteht ein neuer Satz von Klassen zur Repräsentation der Sprachkonstrukte.

[1] *Variante* und *Sprachkonstrukt* wird im folgenden meist synonym verwendet.

```
class Lambda extends Tree {
   Variable v;
   Tree     body;
   Lambda(Variable v, Tree b) {
      this.v = v; body = b;
   }
}
class Apply extends Tree {
   Tree fn, arg;
   Apply(Tree f, Tree a) {
      fn = f; arg = a;
   }
   Tree eval(Env env) {
      Tree fun = fn.eval(env);
      if (fun instanceof Lambda) {
         Lambda f = (Lambda)fun;
         return f.body.eval(new Env(f.v.name, arg.eval(env), env));
      }
      else
         throw new Error("function expected");
   }
}
```

Programm 2.2: Neue Sprachkonstrukte für den objektorientierten Interpreter

2. Es muß sichergestellt werden, daß jeder Klient der auf die Objektstruktur zugreift, auch Objekte der Unterklassen erzeugt. Dies ist beispielsweise nur durch den konsequenten Einsatz von geeigneten Entwurfsmustern zur Objektinstanziierung, wie z.B. dem *AbstractFactory*-Muster [G$^+$95], möglich.

3. Die neuen Klassen haben keinen gemeinsamen Typ der die gesamte Signatur, d.h. alle anwendbaren Operationen wiederspiegelt. In Java kann dieses Problem nur durch die Verwendung von *Interfaces* gelöst werden. Programm 2.3 zeigt die Vorgehensweise. Für Sprachen die keine Interfaces und keine Mehrfachvererbung unterstützen, ist es nicht möglich, einen erweiterten gemeinsamen Typ anzugeben.

4. Wie Abschnitt 4.4.5 noch zeigen wird, ist der objektorientierte Ansatz unbrauchbar für erweiterbare Übersetzer, da sich der Typ einer Datenstruktur ändert, wenn Operationen modifiziert oder neu definiert werden. Wird hier eine Operation, z.B. durch das Überschreiben einer Methode modifiziert, so kann auf die alte Operation als Ganzes nicht mehr zugegriffen werden. Es muß sich zum Wechseln einer Operation die Identität der Objektstruktur ändern. Diese Konsequenz ist weder intuitiv einsehbar, noch erlaubt sie effiziente Lösungen. Zum anderen ist es oftmals schwer möglich, die Identität eines Objekts ohne weiteres zu ändern.

Ein weiterer Nachteil des Interpreter Entwurfsmusters besteht darin, daß die Definition einer Operation über viele Klassen hinweg verteilt und nicht als ganzes an einer Stelle vorzufinden ist. Stattdessen ist der Code mit dem Code vieler anderer Operationen vermischt. Das kann bei vielen einzelnen Operationen sehr verwirrend sein. Außerdem erschwert dies das Verständnis für die Funktionalität einer Operation und macht das Ändern von Operationen wesentlich komplizierter.

```
interface Tree {
   int toInt();
   Tree eval(Env env);
}
class Number implements Tree {
   int value;
   Number(int v) { value = v; }
   int toInt() { return value; }
   Tree eval(Env env) { return this; }
}
class Variable implements Tree {
   String name;
   Variable(String n) { name = n; }
   int toInt() {
      throw new Error("number expected");
   }
   Tree eval(Env env) {
      return env.lookup(name);
   }
}
...
```

```
interface ExtTree extends Tree {
   void print();
}
class ExtNumber extends Number
                implements ExtTree {
   void print() { ... }
}
class ExtVariable extends Variable
                  implements ExtTree {
   void print() { ... }
}
...
```

Programm 2.3: Neue Operation `print` für die objektorientierte Implementierung

2.2 Visitors

Ein klassisches Entwurfsmuster, das man für die gleichen Zwecke wie das Interpreter-Muster einsetzen kann, welches aber konträre Eigenschaften besitzt, ist der *Visitor* [G+95]. Visitors dienen zur Repräsentation von Operationen, die auf die Elemente einer rekursiven Objektstruktur angewendet werden sollen. Sie erlauben es, neue Operationen zu vereinbaren, ohne die Klassen zu verändern, auf denen die Operation definiert ist.

2.2.1 Klassische Visitors

Ein Visitor ist ein Objekt, das eine Operation kapselt. Er definiert für alle Varianten, auf die die Operation angewendet werden darf, eine eigene `visit`-Methode, die die Operation speziell für Objekte dieser Variante implementiert. Jede Variante muß ihrerseits eine `accept`-Methode zur Verfügung stellen, die beliebige Visitor-Objekte akzeptiert und die zu diesem Sprachkonstrukt gehörige Routine des Visitor-Objekts aufruft. Programm 2.4 zeigt das Visitor-Framework für die Beispielsprache.

Wie beim Interpreter-Muster wird für jedes Sprachkonstrukt eine eigene konkrete Klasse definiert. Diese Klassen besitzen als einzige Methode die besagte `accept`-Methode. Ein Visitor wird in Java am besten durch ein Interface beschrieben, das die überladenen `visit`-Methoden für alle Varianten aufzählt. In 2.4 wird zudem noch eine abstrakte Visitor-Oberklasse `DefaultVisitor` definiert, die alle `visit`-Methoden auf einen Standardfall `defaultVisit` abbildet [Nor96]. Damit kann man wie beim Interpreter-Entwurfsmuster für alle Varianten ein Standardverhalten vordefinieren. Leitet man einen konkreten Visitor von `DefaultVisitor` ab, muß man nur noch die `visit`-Methoden überschreiben, bei denen ein individuelles Verhalten benötig wird. Besonders bei komplexen

```
abstract class Tree {
   void accept(Visitor v) { v.visit(this); }

   static class Number extends Tree {
      int value;
      Number(int v) { value = v; }
      void accept(Visitor v) { v.visit(this); }
   }
   static class Variable extends Tree {
      String name;
      Variable(String n) { name = n; }
      void accept(Visitor v) { v.visit(this); }
   }
   static class Plus extends Tree {
      Tree left, right;
      Plus(Tree l, Tree r) {left = l; right = r;}
      void accept(Visitor v) { v.visit(this); }
   }
   static class Times extends Tree {
      Tree left, right;
      Times(Tree l, Tree r){left = l; right = r;}
      void accept(Visitor v) { v.visit(this); }
   }
}
```

```
interface Visitor {
   void visit(Tree.Number tree);
   void visit(Tree.Variable tree);
   void visit(Tree.Plus tree);
   void visit(Tree.Times tree);
}

abstract class DefaultVisitor implements Visitor{
   public void visit(Tree.Number tree) {
      defaultVisit(tree);
   }
   public void visit(Tree.Variable tree) {
      defaultVisit(tree);
   }
   public void visit(Tree.Plus tree) {
      defaultVisit(tree);
   }
   public void visit(Tree.Times tree) {
      defaultVisit(tree);
   }
   abstract void defaultVisit(Tree tree);
}
```

Programm 2.4: Typdeklaration und Visitor-Framework

Sprachen mit vielen Sprachkonstrukten ist ein solcher *Default Visitor* sehr hilfreich, da in den meisten Fällen bei einer Operation nur für wenige Varianten ein eigenes Verhalten benötigt wird und alle restlichen durch eine einzige Standardprozedur bearbeitet werden können. Müßte man für viele Varianten die Standardprozedur einzeln implementieren, würde auch die Lesbarkeit der Operation stark darunter leiden. Programm 2.5 zeigt die Visitorimplementierungen für die beiden Operationen `toInt` und `eval`.

Neben den `visit`-Methoden besitzt jeder Visitor aus Programm 2.5 auch noch eine Methode, über die man die eigentliche Operation aufrufen kann. Diese Methode speichert die Argumente eines Operationsaufrufs in Instanzvariablen des Visitor-Objekts und wendet anschließend den Visitor auf die betreffende Objektstruktur an. Ohne diese zusätzliche Methode wäre es nur schwer möglich, Argumente an einen Visitor zu übergeben und ein Resultatswert zurückzuliefern. Hätte man in Java parametrische Polymorphie [OW97, B^+98], könnte man das Problem zumindest für Spezialfälle – d.h. Operationen mit einer festen Anzahl von Parametern – durch ein generisches Visitor-Interface lösen.

2.2.2 Erweiterung eines Visitors

Neue Operationen implementiert man einfach in neuen Visitor-Klassen. Für die Erweiterung der Beispielsprache um λ-Abstraktionen ist es allerdings notwendig, daß man neue Sprachkonstrukte definiert, also den Datentyp um neue Varianten erweitert und bestehende Visitors entsprechend anpaßt. Programm 2.6 zeigt die nötigen Ergänzungen.

Bereits das Deklarieren neuer Datentyp-Varianten ist problematisch, da die zu implementierenden `accept`-Methoden lediglich ein nicht-erweitertes Visitorobjekt verlangen. In

```
class Evaluator extends DefaultVisitor {
   Env   env;    // Argument
   Tree  res;    // Resultat
   ToInt ti;
   Evaluator(ToInt ti) {
      this.ti = ti;
   }
   public void visit(Tree.Variable tree) {
      res = env.lookup(tree.name);
   }
   public void visit(Tree.Plus tree) {
      res = new Tree.Number(
         ti.toInt(eval(tree.left, env)) +
         ti.toInt(eval(tree.right, env)));
   }
   public void visit(Tree.Times tree) {
      res = new Tree.Number(
         ti.toInt(eval(tree.left, env)) *
         ti.toInt(eval(tree.right, env)));
   }
   public void defaultVisit(Tree tree) {
      res = tree;
   }
   Tree eval(Tree tree, Env env) {
      Env old = this.env; this.env = env;
      tree.accept(this);
      env = old; return res;
   }
}
```

```
class ToInt extends DefaultVisitor {
   int res;
   public void visit(Tree.Number tree) {
      res = tree.value;
   }
   public void defaultVisit(Tree tree) {
      throw new Error("number expected");
   }
   int toInt(Tree tree) {
      tree.accept(this);
      return res;
   }
}
```

Programm 2.5: Als Visitor implementierter Interpreter

```
abstract class ExtendedTree extends Tree {
   static class Lambda extends ExtendedTree {
      Variable v;
      Tree     body;
      Lambda(Variable x, Tree b){v = x;body = b;}
      void accept(Visitor v) {
         ((ExtendedVisitor)v).visit(this);
      }
   }
   static class Apply extends ExtendedTree {
      Tree fn, arg;
      Apply(Tree f, Tree a) { fn = f; arg = a; }
      void accept(Visitor v) {
         ((ExtendedVisitor)v).visit(this);
      }
   }
}

interface ExtendedVisitor extends Visitor {
   void visit(ExtendedTree.Lambda tree);
   void visit(ExtendedTree.Apply tree);
}
```

```
class ExtendedToInt extends ToInt
                    implements ExtendedVisitor {
   public void visit(ExtendedTree.Lambda tree) {
      defaultVisit(tree);
   }
   public void visit(ExtendedTree.Apply tree) {
      defaultVisit(tree);
   }
}
class ExtendedEvaluator extends Evaluator
                     implements ExtendedVisitor{
   ExtendedEvaluator(ExtendedToInt ti) {
      super(ti);
   }
   public void visit(ExtendedTree.Lambda tree) {
      defaultVisit(tree);
   }
   public void visit(ExtendedTree.Apply tree) {
      Tree    fun = eval(tree.fn, env);
      if (fun instanceof ExtendedTree.Lambda) {
         ExtendedTree.Lambda f =
            (ExtendedTree.Lambda)fun;
         res = eval(f.body, new Env(f.v.name,
            eval(tree.arg, env), env));
      }
      else
         throw new Error("function expected");
   }
}
```

Programm 2.6: Erweiterte Visitors

diesem Objekt müssen aber gar keine `visit`-Methoden für die neuen Varianten definiert sein. Deswegen ist es nötig, in den `accept`-Methoden neuer Varianten einen Typecast auf einen erweiterten Visitor einzufügen. Wendet man einen „alten“ Visitor auf eine Datenstruktur an, die neue Varianten referenziert, so erhält man einen Laufzeitfehler, da der Typecast fehlschlägt. Man verliert also beim Erweitern des Datentyps Typsicherheit und muß selbst sicherstellen, daß man bei einer einzigen Typerweiterung ausnahmslos alle bestehenden Visitor-Klassen durch Unterklassenbildung um die neuen `visit`-Methoden ergänzt. Wie man am Visitor `ExtendedToInt` sehen kann, ist dies selbst dann nötig, wenn die neuen Varianten bei einem Default Visitor durch den Standardfall abgedeckt werden. Dieser erspart nur für die initialen Datentyp-Varianten Arbeit.

Hat man beim Entwurfsmuster Interpreter für das Erweitern um eine einzige Operation alle konkreten Klassen erweitern müssen, so ist das bei den Visitors nun genau umgedreht: Fügt man hier eine einzige Variante zum Datentyp hinzu, muß man alle Operationen, d.h. Visitor-Klassen, erweitern.

Zusammenfassend läßt sich feststellen, daß Visitors eine lokale Definition von Operationen erlauben, so daß die Definition einer Operation nicht mehr über mehrere Klassen verteilt wird, wie das beim Entwurfsmuster Interpreter der Fall ist. Visitors ermöglichen auf einfache Art und Weise, neue Operationen für einen Datentyp zu schreiben. Umgekehrt sind Erweiterungen des Datentyps ziemlich kompliziert, aufwendig und unsicher. Weiterhin besitzt das eigentliche Entwurfsmuster kein Konzept, zusätzliche Argumente an eine Operation zu übergeben bzw. einen Wert zurückzuliefern. Ein entscheidender Nachteil, was das Laufzeitverhalten angeht, ist der mehrfache Methodenaufruf. Mindestens ein Dispatch für das Datentyp-Objekt und das Visitor-Objekt sind nötig. Für das obige Beispiel finden pro Operationsaufruf sogar drei Methodenaufrufe statt.

2.2.3 Alternative Realisierungen erweiterbarer Visitors

Interessanterweise haben das Problem erweiterbarer Visitors in den letzten Monaten mehrere Leute gleichzeitig und unabhängig voneinander aufgegriffen. Alle in den folgenden beiden Abschnitten vorgestellten erweiterbaren Visitor-Entwurfsmuster wurden zwischen Januar und März 1998 veröffentlicht. Die Ansätze erklären erweiterbare Visitors ausnahmslos im Kontext von Java.

2.2.3.1 Entwurfsmuster *Extensible Visitor*

In [KFF98] wird ein zusammengesetztes Entwurfsmuster für erweiterbare Visitors beschrieben. Diese Lösung entspricht weitgehend dem zuvor vorgestellten Erweiterungsprinzip. Argumente werden auch bei dieser Lösung in Instanzvariablen des Visitorobjekts gespeichert. Bei jedem Aufruf der Operation wird hier aber im allgemeinen ein neues Visitor-Objekt über eine Factory-Methode erzeugt und die Operation über

```
class Walkabout {
   void visit(Object v) {
      if (v != null)
         if (this has a public visit method for the type of v)
            this.visit((class of v)v);
         else if (v is not of primitive type)
            for (each field f of v)
               this.visit(v.f);
   }
}
```

Programm 2.7: Oberklasse generischer Visitors

dieses neue Objekt aufgerufen. Damit entfällt das Zwischenspeichern der Operations-Argumente in lokalen Variablen, wie das beispielsweise für den `Evaluator`-Visitor im Beispielprogramm 2.5 gezeigt wurde. Allerdings zieht jeder Operationsaufruf damit insgesamt vier Methodenaufrufe nach sich.[2] Außerdem entsteht bei jedem Operationsaufruf unnötigerweise ein neues Objekt. Da das Protokoll für erweiterbare Visitors im allgemeinen ziemlich kompliziert und fehleranfällig ist, wurde für das Entwurfsmuster *Extensible Visitor* eine Metasprache entworfen, die die Spezifikation von Instanzen erweiterbarer Visitors erleichtern soll.

In Zusammenhang mit einem objektorientierten Übersetzer-Framework für Java wird in [GH98, Gag98] ein konkurrierendes Entwurfsmuster für erweiterbare Visitors beschrieben. Dieses Muster ist ähnlich zu dem von [KFF98]. Es ist insbesondere mit den gleichen Problemen behaftet. Allerdings erlaubt dieses Entwurfsmuster auf eine flexiblere Art und Weise, erweiterte Visitor-Schnittstellen aus existierenden zusammenzusetzen. Auch bei diesem Ansatz muß der Programmierer nicht das aufwendige Visitor-Framework per Hand implementieren. Es wird aus einer Spezifikation der Sprache automatisch generiert.

2.2.3.2 Generische Visitors

Palsberg und Jay lösen das Problem, daß zur Definition eines klassischen Visitors die Klassen aller Objekte, auf denen operiert wird, bekannt und damit fest sind, durch einen völlig anderen Ansatz. Inspiriert durch Shape Polymorphismus aus dem funktionalen Programmieren [Jay95] beschreiben sie in [PJ97] generische Visitors. Diese sogenannten *Walkabouts* benötigen nicht mehr das übliche Visitor-Framework. An die Stelle des Visitor-Interfaces tritt eine Oberklasse `Walkabout`, von der alle generischen Visitors abgeleitet werden. Programm 2.7 zeigt eine Pseudo-Code-Formulierung dieser Klasse.

Die `Walkabout`-Klasse besitzt lediglich eine einzige `visit`-Methode für beliebige Objekte. Diese untersucht, ob es im generischen Visitor eine für das übergebene Objekt spezialisierte `visit`-Methode gibt und ruft diese gegebenenfalls auf. Wird keine solche Methode gefunden, wird auf alle Instanzvariablen des übergebenen Objekts der generische Visitor

[2] Neben der `accept`- und der `visit`-Methode ist jeweils noch eine Factory-Methode und ein Konstruktor aufzurufen.

angewendet. Die Java-Implementierung in [PJ97] benutzt die *Reflection*-Bibliothek, um die Struktur einer anonymen Klasse zu untersuchen.

Die Idee, das Untersuchen eines Objekts und die Operation auf diesem Objekt zu trennen, ermöglicht maximale Flexibilität was die Anwendung und Erweiterbarkeit generischer Visitors betrifft. Allerdings zeigen die Benchmarks in [PJN98], daß man bei dieser Lösung, verglichen mit herkömmlichen Visitors, eine um zwei Größenordnungen größere Laufzeit in Kauf nehmen muß. Das macht Walkabouts für die Praxis irrelevant.

2.2.3.3 Programmiersprachenunterstützung für Visitors

Es gab bisher bereits einige Ansätze, durch spezielle Programmiersprachkonstrukte das Schreiben von Visitors zu vereinfachen bzw. Erweiterungen einfacher zu ermöglichen. Baumgartner, Läufer und Russo schlagen Visitors basierend auf *Multi-Methoden* vor [BLR96]. Multi-Methoden sind Methoden, bei denen der Methodendispatch dynamisch über alle Argumente durchgeführt wird. Damit entfällt der wechselseitige Aufruf zwischen `accept`- und `visit`-Methoden beim üblichen Visitor. Boyland und Castagna integrierten Multi-Methoden in Java [BC97]. Ein Multi-Methoden-Dispatch wird hier unter Verwendung des `instanceof`-Operators implementiert.

In einer Sprache die direkt ein *Typeswitch*-Konstrukt zur Verfügung stellt, werden Visitors nicht benötigt. Dieses Konstrukt erlaubt es direkt, Operationen in funktionalem Stil zu formulieren. Bei beiden Vorschlägen ist fraglich, wie effizient es überhaupt möglich ist, diese in einer Programmiersprache zu realisieren.

2.3 Typschalter

Visitors erlauben es zwar mit objektorientierten Mitteln Daten und Funktionen getrennt zu deklarieren und zu erweitern, sind aber nicht sonderlich flexibel einsetzbar. In Übersetzern kommen recht oft kleine Fallunterscheidungen über die Varianten eines Datentyps vor. Hierfür jedesmal einen eigenen Visitor zu schreiben, würde zu einer unüberblickbaren Flut von Visitor-Klassen führen. Dies würde die Verständlichkeit des Codes stark beeinträchtigen. Deswegen werden bei vielen Übersetzern die Varianten der Strukturbaumrepräsentation mit *Tags* versehen, über welche man effizient den Typ eines Objekts bestimmen kann. Außerdem bieten viele Programmiersprachen eine `switch`-Anweisung an, mittels der es dann relativ einfach ist, einen *Typschalter* über die Varianten eines Datentyps zu implementieren. In Programm 2.8 wird ein Interpreter nach diesem Muster aufgebaut.

In der abstrakten Oberklasse `Tree` wird für jedes Sprachkonstrukt ein Tag definiert. Über die `tag`-Variable eines Objekts kann dann der dynamische Variantentyp abgefragt werden. Operationen kann man nun ganz einfach durch Methoden implementieren, die über eine `switch`-Anweisung eine Fallunterscheidung über alle Varianten durchführen.

```
abstract class Tree {
   final static int NUMBER   = 0;
   final static int VARIABLE = 1;
   final static int PLUS     = 2;
   final static int TIMES    = 3;
   final int tag;

   Tree(int tag) {
      this.tag = tag;
   }
   static class Number extends Tree {
      int        value;
      Number(int value) {
         super(NUMBER);
         this.value = value;
      }
   }
   static class Variable extends Tree {
      String  name;
      Variable(String name) {
         super(VARIABLE);
         this.name = name;
      }
   }
   static class Plus extends Tree {
      Tree    left, right;
      Plus(Tree l, Tree r) {
        super(PLUS);
        left = l; right = r;
      }
   }
   static class Times extends Tree {
      Tree    left, right;
      Times(Tree l, Tree r) {
         super(TIMES);
         left = l; right = r;
      }
   }
}
```

```
class Evaluator {
   int toInt(Tree tree) {
      switch (tree.tag) {
         case Tree.NUMBER:
            return ((Tree.Number)tree).value;
         default:
            throw new Error("number expected");
      }
   }
   Tree eval(Tree tree, Env env) {
      switch (tree.tag) {
         case Tree.VARIABLE:
            return env.lookup(
                     ((Tree.Variable)tree).name);
         case Tree.PLUS:
            Tree.Plus plus = (Tree.Plus)tree;
            return new Tree.Number(
                  toInt(eval(plus.left, env)) +
                  toInt(eval(plus.right, env)));
         case Tree.TIMES:
            Tree.Times times = (Tree.Times)tree;
            return new Tree.Number(
                  toInt(eval(times.left, env)) *
                  toInt(eval(times.right,env)));
         default:
            return tree;
      }
   }
}
```

Programm 2.8: Implementierung eines Interpreters mit Typschaltern

In jedem Fall der `switch`-Anweisung muß dann als erstes ein Typecast auf den tatsächlichen Variantentyp vorgenommen werden, bevor auf die einzelnen Felder der Variante zugegriffen werden kann.

Programm 2.9 zeigt, wie man Programm 2.8 erweitern kann, daß λ-Abstraktionen unterstützt werden. In der Oberklasse der neuen Datentypvarianten werden die Tags für die neuen Sprachkonstrukte vergeben. Operationen lassen sich recht elegant durch Vererbung erweitern. Für Klassen die Operationen definieren, werden Unterklassen angelegt, in denen die zu modifizierenden Methoden einfach überschrieben werden. Da eine überschriebene Methode auch in der überschreibenden Methode aufgerufen werden kann, ist es möglich, die alte Operation flexibel wiederzuverwenden. Die Erweiterung von `eval` zur Unterstützung der beiden neuen Sprachkonstrukte ist ein Beispiel dafür, wie man typischerweise den Definitionsbereich einer Operation erweitert: Man macht eine Fallunterscheidung in der man die neuen Varianten behandelt und verweist die übrigen Varianten an die bisherige Definition. Die `toInt`-Methode muß nicht überschrieben werden,

```
abstract class ExtendedTree extends Tree {
   final static int LAMBDA   = 4;
   final static int APPLY    = 5;
   ExtendedTree(int tag) {
      super(tag);
   }
   static class Lambda extends ExtendedTree {
      Variable  v;
      Tree      body;
      Lambda(Variable v, Tree b) {
         super(LAMBDA);
         this.v = v; this.body = b;
      }
   }
   static class Apply extends ExtendedTree {
      Tree fn, arg;
      Apply(Tree fn, Tree arg) {
         super(APPLY);
         this.fn = fn; this.arg = arg;
      }
   }
}
class ExtendedEvaluator extends Evaluator {
   Tree eval(Tree tree, Env env) {
      switch (tree.tag) {
         case ExtendedTree.APPLY:
            ExtendedTree.Apply apply = (ExtendedTree.Apply)tree;
            Tree               fun   = eval(apply.fn, env);
            switch (fun.tag) {
               case ExtendedTree.LAMBDA:
                  ExtendedTree.Lambda lambda = (ExtendedTree.Lambda)fun;
                  return eval(lambda.body, new Env(lambda.v.name, eval(apply.arg, env), env));
               default:
                  throw new Error("function expected");
            }
         default:
            return super.eval(tree, env);
      }
   }
}
```

Programm 2.9: Erweiterung der Typschalter-Lösung

da für die beiden neuen Konstrukte keine spezielle Behandlung nötig wird. Sie fallen automatisch in den `default`-Fall der ursprünglichen Fallunterscheidung.

Im Gegensatz zu dem Ansatz mit Visitors, ist es bei dieser Lösung möglich, sowohl Daten als auch Operationen unabhängig voneinander zu erweitern. Typschalter sind außerdem flexibler einsetzbar als Visitors: Sie können an jeder Stelle eines Programms eingebaut werden. Allerdings dürfte die Typschalter-Lösung wesentlich fehleranfälliger sein. Die Tags müssen von Hand verwaltet werden und auch das Typschalter-Muster ist kompliziert zu implementieren. Durch die ständig benötigten Typecasts unterlaufen Typschalter unvermeidbar das Typsystem einer objektorientierten Sprache. Implementationsfehler machen sich deswegen oft erst zur Laufzeit bemerkbar. Der folgende Abschnitt zeigt, wie eine erweiterbare Form von algebraischen Datentypen dazu verwendet werden kann, erweiterbare Interpreter im hier vorgestellten Stil ohne Umgehung des Typsystems zu schreiben.

```
class Tree {
   case Number(int value);
   case Variable(String name);
   case Plus(Tree left, Tree right);
   case Times(Tree left, Tree right);
}

final class Env {
   case Empty;
   case Bind(String name, Tree value, Env next);
   Tree lookup(String ident) {
      switch (this) {
         case Empty:
            throw new Error(ident + " unbound");
         case Bind(String n, Tree val, Env next):
            if (ident.equals(n))
               return val;
            else
               return next.lookup(ident);
      }
   }
}
```

```
class Evaluator {
   int toInt(Tree tree) {
      switch (tree) {
         case Number(int value):
            return value;
         default:
            throw new Error("number expected");
      }
   }
   Tree eval(Tree tree, Env env) {
      switch (tree) {
         case Variable(String name):
            return env.lookup(name);
         case Plus(Tree left, Tree right):
            return Tree.Number(
               toInt(eval(left, env)) +
               toInt(eval(right, env)));
         case Times(Tree left, Tree right):
            return Tree.Number(
               toInt(eval(left, env)) *
               toInt(eval(right, env)));
         default:
            return tree;
      }
   }
}
```

Programm 2.10: Implementierung eines Interpreters mit algebraischen Typen

2.4 Funktionale Interpreter

2.4.1 Interpreter in funktionalen Sprachen

Beim funktionalen Programmieren werden Daten und Funktionen getrennt voneinander definiert. Zur Repräsentation der abstrakten Syntax einer Sprache verwendet man üblicherweise *algebraische Datentypen*. Die Varianten eines Datentyps werden hier durch eine Menge von Konstruktoren beschrieben. Funktionen bilden die Varianten auf Ergebniswerte ab. Über Pattern Matching lassen sich dabei einfach die einzelnen Fälle unterscheiden.

Programm 2.10 zeigt eine funktionale Implementierung der Beispielsprache. Da Java als objektorientierte Sprache keine algebraischen Typen bietet, wurde für Programm 2.10 Pizza [OW97] verwendet. Die Varianten eines algebraischen Typs vereinbart man in Pizza durch `case`-Deklarationen in einer *algebraischen Klasse*. Funktionen lassen sich beispielsweise durch gewöhnliche Methoden implementieren. Pattern Matching wird in Pizza durch die `switch`-Anweisung unterstützt.

Da Operationen wie in der Typschalter-Lösung nichts anderes als gewöhnliche Methoden sind, ist es äußerst einfach, neue Operationen zu schreiben. Unglücklicherweise ist es jetzt aber unmöglich, neue Konstrukte zur Syntax einer Sprache hinzuzufügen, ohne bestehenden Code zu verändern. Hierfür gibt es zwei Ursachen. Zum einen muß der Datentyp um neue Varianten ergänzt werden. Dies ist jedoch gewöhnlich nicht möglich,

```
class ExtendedTree extends Tree {
   case Lambda(Variable v, Tree body);
   case Apply(Tree fn, Tree arg);
}
class ExtendedEvaluator extends Evaluator {
   Tree eval(Tree tree, Env env) {
      switch ((ExtendedTree)tree) {
         case Apply(Tree fn, Tree arg):
            switch ((ExtendedTree)eval(fn, env)) {
               case Lambda(Tree.Variable v, Tree body):
                  return eval(body, Env.Bind(v.name, eval(arg, env), env));
               default:
                  throw new Error("function expected");
            }
         default:
            return super.eval(tree, env);
      }
   }
}
```

Programm 2.11: Erweiterung des Interpreters mittels erweiterbarer algebraischer Typen

da die meisten funktionalen Programmiersprachen – Pizza eingeschlossen – es nicht erlauben, algebraische Typen zu erweitern. Zum anderen müssen bestehende Operationen so geändert werden können, daß sie die neuen Sprachkonstrukte unterstützen. Einmal definierte Funktionen können jedoch im nachhinein in funktionalen Sprachen bestenfalls um neue Fälle erweitert, aber ansonsten nicht abgeändert werden. Es ist auch hier nötig den Quelltext umzuschreiben. Im Gegensatz zum objektorientierten Ansatz von 2.1 unterstützt die funktionale Lösung also lediglich die Erweiterbarkeit um neue Operationen, nicht jedoch die Erweiterbarkeit von Datentypen.

2.4.2 Erweiterung funktionaler Interpreter

Mit Objekttypen und Vererbung werden Interpreter in objektorientiertem Stil geschrieben, wohingegen algebraische Typen und Pattern Matching die korrespondierenden Mechanismen auf der funktionalen Seite sind. Eine Sprache wie Pizza hat jedoch beides. Hätte man hier erweiterbare algebraische Typen, könnte man ganz einfach auch Funktionen realisieren, die man nachträglich erweitern bzw. modifizieren kann.

Programm 2.11 zeigt das Prinzip. Für dieses Beispiel wird vorausgesetzt, daß algebraische Datentypen erweiterbar sind. `ExtendedTree` ist demnach ein Typ, der alle Varianten vom algebraischen Typ `Tree` erbt und zusätzlich noch Konstrukte für λ-Abstraktion und -Applikation definiert. Das Erweitern der Funktionen ist jetzt wie bei den Typschaltern über Vererbung möglich. Klassen, die Operationen wie `eval` und `toInt` definieren, haben in unserem Fall die Funktion von Modulen. Sie kapseln zusammengehörige Operationen auf einem bestimmten Datentyp. Durch Unterklassenbildung ist es möglich, einzelne Methoden zu überschreiben, um sie an die neuen Sprachkonstrukte anzupassen. Da man beim Überschreiben stets auch noch die überschriebene Methode referenzieren kann, ist es möglich, die bisherige Funktionsdefinition flexibel wiederzuverwenden. Die

Erweiterung bzw. Veränderung von Funktionen wird also auf die gleiche Weise wie bei der Typschalter-Lösung realisiert. An die Stelle der Typschalter tritt jetzt das Pattern Matching.

Es sei darauf hingewiesen, daß auch das Programm 2.11 einen Typecast enthält, der ein Pattern Matching über den erweiterten Typ ermöglicht. Dieser wird allerdings nur wegen des invarianten Überschreibens von Methoden in Java benötigt: Überschreibende Methoden müssen die gleiche Signatur wie die überschriebenen Methoden besitzen. Über einen Ausdruck des Typs `ExtendedTree` sind Pattern Matching-Ausdrücke selbstverständlich ohne irgendwelche Typecasts möglich. Visitors und Typschalter sind dagegen stets auf Casts angewiesen.

Die hier vorgestellte Lösung stellt eine Kombination aus dem reinen objektorientierten und dem funktionalen Ansatz dar. Erweiterbare algebraische Typen und Pattern Matching werden dazu benutzt, die Datenstruktur zu repräsentieren und Fallunterscheidungen durchführen zu können, wohingegen Klassen und Vererbung dazu verwendet werden, erweiterbare Module höherer Ordnung zu implementieren. Dieses Verfahren synthetisiert von beiden Ansätzen jeweils die Vorteile. Es erlaubt auf sichere und einfache Art und Weise, sowohl Datentyp als auch die Menge der Operationen unabhängig voneinander zu erweitern, sowie bestehende Operationen zu modifizieren, ohne bestehenden Code zu verändern.

2.5 Zusammenfassung

Objekttypen und Vererbung finden im funktionalen Ansatz nur zur Implementierung von erweiterbaren Modulen höherer Ordnung Verwendung. Die Interpreter selbst werden funktionaler Entwurfsprinzipien folgend implementiert. Bereits der objektorientierte Ansatz über *Visitors* hat konzeptionell einer funktionalen Architektur entsprochen. Daten und Operationen wurden getrennt voneinander vereinbart. Allerdings wurden hier sowohl Daten als auch die Operationen uniform durch Objekte dargestellt. Dies bedeutete sowohl für die Daten- als auch für die Operationsdefinitionen, daß man diese in einer durch das Entwurfsmuster vorgegebenen Schablone notieren muß. Damit wird die Produktivität beim Programmieren wesentlich eingeschränkt. Außerdem führt der Zwang, alle Operationen als Visitor implementieren zu müssen, fast zwangsläufig zu einer großen Anzahl von Visitor-Klassen, die oftmals nur sehr kleine, einfache Operationen implementieren.

Die *Typschalter*-Lösung ist sehr verwandt mit der zuvor vorgestellten funktionalen Realisierung, besitzt allerdings als große Schwäche, daß Implementierung und Erweiterung aufwendig, unsicher und fehleranfällig sind. Es muß ein Protokoll implementiert werden, bei dem bereits in der unerweiterten Version ständig Typecasts benötigt werden. Das Typsystem wird auf diese Weise bei Typschaltern quasi außer Gefecht gesetzt. Außerdem führen die vielen Casts auch dazu, daß der Code verglichen mit Programm 2.10, in welchem algebraische Typen verwendet wurden, relativ schlecht zu lesen ist.

Wie die Typschalter-Lösung, ist die rein *funktionale Realisierung* mit erweiterbaren algebraischen Typen flexibler als der Visitor-basierte Ansatz und bietet durch das Pattern Matching wesentlich mehr Freiheiten. Sowohl Datentypen als auch Operationen lassen sich auch hier frei und unabhängig voneinander erweitern. Typ- und Operationsdefinitionen liegen lokal vor und sind deswegen leichter verständlich als bei einer verteilten Definition über mehrere Klassen hinweg. Pattern Matching-Ausdrücke tragen wesentlich zur Verständlichkeit eines Programms bei. Schon alleine die Kürze und Lesbarkeit der Programme 2.10 und 2.11 überzeugt, daß der funktionale Ansatz über algebraische Typen und Pattern Matching, den objektorientierten Lösungen mit Visitors und Typschaltern vorzuziehen ist. Zur Implementierung der komplizierten Protokolle für erweiterbare Visitors finden oftmals Meta-Sprachen Verwendung. Da algebraische Typen einfache und intuitive Problemlösungen zulassen, ist man hier auf ähnliche Hilfsmittel nicht angewiesen. Ein wichtiger Punkt ist auch, daß es bei der funktionalen Lösung nicht nötig wird, das Typsystem durch Typecasts zu unterlaufen. Schließlich wird Kapitel 3 zeigen, daß sich erweiterbare algebraische Typen und Pattern Matching relativ effizient implementieren lassen. Sie sind also auch für zeitkritische Anwendungen geeignet.

2.6 Benchmarks

Alle in den vorangegangenen Abschnitten dieses Kapitels vorgestellten Entwurfsstrategien für erweiterbare Interpreter wurden implementiert. Als Übersetzer wurde der im Rahmen dieser Arbeit entwickelte Java-Übersetzer verwendet. Um die Effizienz der einzelnen Ansätze gegenüberstellen zu können, wurden die Interpreter wiederholt auf einen größeren Testausdruck der Beispielsprache angewendet und die dafür benötigte Berechnungszeit gemessen.[3] Um die Effizienz einer Lösung unabhängig von den Kosten einer Erweiterung messen zu können, wurden von jedem Ansatz zwei Versionen angefertigt. Die erste Version implementiert die gesamte Beispielsprache ohne daß der zugehörige Erweiterungsmechanismus verwendet wird. In der zweiten Version wird zunächst der Interpreter für die Basissprache entwickelt und anschließend explizit um λ-Abstraktionen erweitert.

Zunächst werden die Versionen betrachtet, bei denen ein vollständiger Interpreter für die gesamte Sprache implementiert wird, ohne daß das zugehörige Erweiterungsprinzip Anwendung findet. Die einzelnen Programme wurden auf drei verschiedenen virtuellen Maschinen jeweils mit und ohne Just-In-Time-Compiler ausgeführt. Konkret wurden das MRJ 2.0 (MacOS Runtime for Java) von Apple Computer, Inc. und die virtuelle Maschine der Firma Metrowerks, Inc. (MW) auf einem Apple Macintosh 7200/90, sowie das JDK 1.2beta3 von Sun Microsystems, Inc. auf einer Sun UltraSPARC 1 eingesetzt. In Tabelle 2.1 sind die Ergebnisse aufgelistet. EAT-Opt. bezeichnet die optimiert über-

[3] Es wird nicht die Gesamtlaufzeit der Programme gemessen, sondern lediglich die Laufzeit des Interpreters. Die Zeiten zum Laden der Klassen oder zur Initialisierung der Objektstrukturen werden also nicht berücksichtigt.

setzte Implementierung über erweiterbare algebraische Typen. Hier wurden vom Übersetzer sichere Typecasts aus dem Bytecode optimiert.[4] Die Laufzeiten sind prozentual bezüglich der objektorientierten Lösung angegeben, da hier kein Vergleich einzelner virtueller Maschinen durchgeführt werden soll. Die Testläufe verschiedener Maschinen sind auch aufgrund der unterschiedlichen Hardware nicht vergleichbar.

Implementierung	Interpreter			JIT		
	MRJ	**MW**	**JDK**	**MRJ**	**MW**	**JDK**
OO	100 %	100 %	100 %	100 %	100 %	100 %
Visitor	206 %	220 %	284 %	159 %	151 %	245 %
Typschalter	124 %	127 %	159 %	123 %	118 %	156 %
EAT	147 %	140 %	168 %	136 %	129 %	160 %
EAT-Opt.	138 %	136 %	155 %	96 %	105 %	117 %

Tabelle 2.1: Benchmarks für die Programmversionen ohne explizite Erweiterung

Wie erwartet ist die Visitor-Lösung allgemein am teuersten, wenngleich die prozentuale Abweichung bezogen auf die übrigen Lösungen hier am meisten variiert. Für die Interpreter-Messungen machen sich vor allem auf der SPARC die insgesamt 3 Methodenaufrufe pro Operationsaufruf gegenüber der OO-Implementierung in Form einer fast dreimal längeren Laufzeit bemerkbar. Die Lösung mit erweiterbaren algebraischen Typen ist hier nur 50 % langsamer. Schaut man sich die Ergebnisse für die Just-In-Time-Compiler an, so ist nun fast kein Unterschied mehr zwischen der OO-Implementierung und der optimiert übersetzten EAT-Implementierung zu messen. Visitors sind aber, vor allem auf der SPARC, immer noch sehr teuer. JIT-Compiler scheinen also vor allem die „`switch`-Lösungen" prozentual stärker zu beschleunigen. Die Typschalter-Ergebnisse liegen wie erwartet bei den Laufzeitmessungen fast immer an zweiter Stelle. Erweiterbare algebraische Typen werden, wie in Kapitel 3.5 erläutert wird, in eine ähnliche Struktur übersetzt. Die kompliziert zu übersetzenden Pattern Matching-Konstrukte sorgen allerdings hier für eine etwas schlechtere Performance. Lediglich bei den JIT-Compilern ist die optimierte Version der erweiterbaren algebraischen Typen auf allen Maschinen besser als die Typschalter-Lösung.

Die Meßwerte aus Tabelle 2.1 spiegeln lediglich die Effizienz der einzelnen Lösungen ohne die Kosten für Erweiterungen wieder. In die Laufzeit für die Benchmarks aus Tabelle 2.2 fließen dagegen die Kosten für eine Erweiterung mit ein. Die Ergebnisse sind unmittelbar mit denen von Tabelle 2.1 vergleichbar, da es sich zum einen um identische Testläufe[5] handelt und zum anderen die objektorientierte Implementierung, bezüglich der die prozentualen Angaben erfolgen, in beiden Fällen identisch ist.

Die für das Erweitern von Visitors notwendigen Typecasts verlangsamen die Testläufe der

[4]Programme, die auf diese Weise optimiert werden, können zwar ohne Probleme auf jeder virtuellen Maschine ausgeführt werden, sie passieren allerdings nicht mehr den Bytecode-Verifier. Details zur Übersetzung von erweiterbaren algebraischen Typen finden sich in Kapitel 3.

[5]Es wurden jeweils die gleichen Testausdrücke interpretiert.

Implementierung	Interpreter			JIT		
	MRJ	**MW**	**JDK**	**MRJ**	**MW**	**JDK**
OO	100 %	100 %	100 %	100 %	100 %	100 %
Visitor	207 %	226 %	285 %	164 %	153 %	254 %
Typschalter	143 %	140 %	196 %	146 %	136 %	179 %
EAT	199 %	164 %	251 %	159 %	152 %	196 %
EAT-Opt.	194 %	162 %	234 %	119 %	129 %	149 %

Tabelle 2.2: Benchmarks für die Programmversionen mit expliziter Erweiterung

erweiterten Visitors nur unwesentlich. Lediglich bei der Ausführung mit JIT-Compiler sind geringe Laufzeiteinbußen erkennbar. Bei den übrigen Implementierungen sieht dies anders aus. Hier wird bei erweiterten Operationen ein zusätzlicher nicht-virtueller Methodenaufruf benötigt, der die alte Routine aus der überschreibenden Methode aus aufruft. Damit verringert sich der Vorsprung dieser Lösungen gegenüber den Visitors, ist aber vor allem bei den JIT-Messungen immer noch ziemlich groß. Tabelle 2.3 legt die Erweiterungskosten der Implementierungen prozentual zur Gesamtlaufzeit der explizit erweiterten Programme offen. Der Erweiterungsanteil ist bei den letzten drei Lösungen, abgesehen von unterschiedlichen Schwankungen bei verschiedenen Maschinen, zumindest größenordnungsmäßig gleich groß.

Implementierung	Interpreter			JIT		
	MRJ	**MW**	**JDK**	**MRJ**	**MW**	**JDK**
OO	0 %	0 %	0 %	0 %	0 %	0 %
Visitor	1 %	3 %	1 %	3 %	1 %	4 %
Typschalter	13 %	9 %	19 %	16 %	13 %	13 %
EAT	26 %	14 %	33 %	14 %	15 %	19 %
EAT-Opt.	29 %	16 %	34 %	19 %	19 %	22 %

Tabelle 2.3: Erweiterungskosten relativ zur Gesamtlaufzeit der Programme

Abschließend werden nochmals die Laufzeiten für die Ergebnisse der Testläufe auf der UltraSPARC mit JDK1.2beta3 in Form von Balkendiagrammen gegenübergestellt. Die Diagramme 2.2 und 2.3 sind dabei so skaliert, daß auch Vergleiche zwischen den beiden Diagrammen möglich sind.

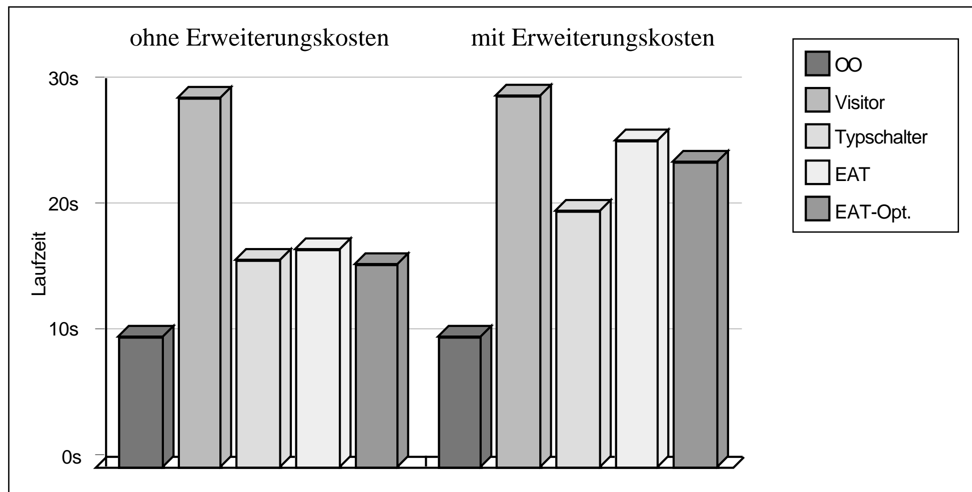


Abbildung 2.2: Benchmarks für die virtuelle Maschine des Solaris JDK

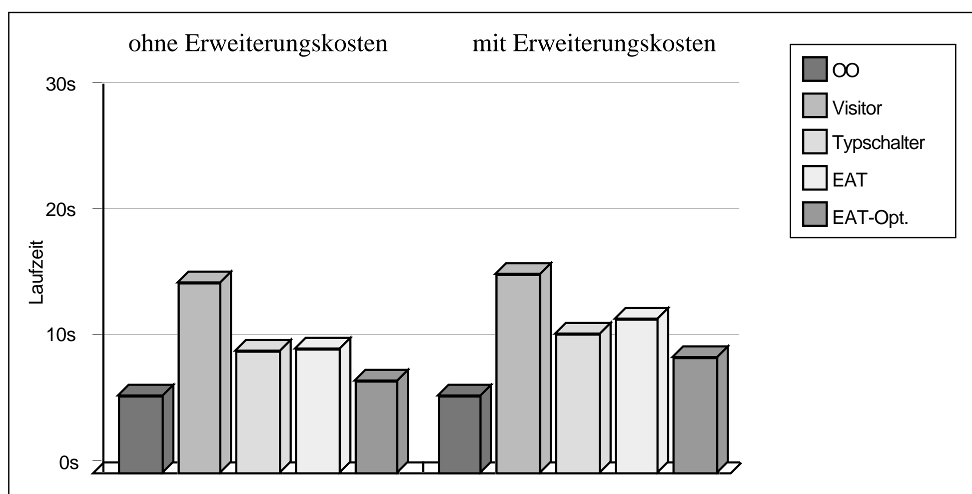


Abbildung 2.3: Benchmarks für den JIT des Solaris JDK

Kapitel 3

Erweiterbare Algebraische Typen

Benutzerdefinierte Typen werden bei den meisten funktionalen Programmiersprachen in Form von algebraischen Datentypen zur Verfügung gestellt. Diese werden teilweise auch *strukturierte Typen* oder *freie Datentypen* genannt. In praktisch allen Programmiersprachen, die algebraische Typen unterstützen, ist es nicht möglich, diese Datentypen zu erweitern. Bisherige Konzepte zur Erweiterung algebraischer Typen besitzen Eigenschaften, die sie für die Praxis nur sehr beschränkt einsetzbar machen. Dieses Kapitel schlägt einen neuartigen, pragmatischen Ansatz für erweiterbare algebraische Datentypen vor, der nicht mit den Nachteilen bisheriger Konzepte behaftet ist. Es wird erläutert, wie diese Datentypen syntaktisch und semantisch in die Programmiersprache Java integriert werden können. Weiterhin wird ein Übersetzungsschema angegeben, mit dessen Hilfe erweiterbare algebraische Typen in reguläres Java übersetzt werden können.

Ein mit algebraischen Typen verbundener Mechanismus zur Formulierung von Fallunterscheidungen ist das Pattern Matching. Es wird diskutiert, wie Pattern Matching für erweiterbare algebraische Typen in Java realisiert werden kann. Abschließend wird ein Verfahren zur effizienten Übersetzung von Pattern Matching-Ausdrücken in reguläre Java-Programme vorgestellt.

3.1 Algebraische Datentypen

Die Notation zur Spezifikation algebraischer Datentypen ist bei den meisten funktionalen Programmiersprachen recht ähnlich. Im folgenden wird eine Syntax verwendet, die sich an Haskell [Pet97] orientiert. Eine Definition eines algebraischen Datentyps A mit zwei Varianten A_1 und A_2 sieht darin folgendermaßen aus:

data A = $\texttt{A}_1$ $T_{1,1} \ldots T_{1,r_1}$
        | $\texttt{A}_2$ $T_{2,1} \ldots T_{2,r_2}$

Hier bezeichnen die $T_{i,j}$ Typen und die $\texttt{A}_\texttt{i}$ stellen Konstruktoren der Stelligkeit r_i dar. Jeder Konstruktor $\texttt{A}_\texttt{i}$ repräsentiert einen Typ $A_i = T_{i,1} \times \ldots \times T_{i,r_i}$. Typ A vereinigt

alle diese Produkttypen, kann also als Summe der A_i geschrieben werden: $A = A_1 + A_2$. Damit sind algebraische Typen nichts anderes als Summen von Produkten [PJ86].

In einigen Sprachen wird ein algebraischer Typ mit genau einem Konstruktor als Produkttyp angesehen. Im Kontext der Erweiterbarkeit bereitet diese Anomalie jedoch Probleme, weswegen im folgenden auch algebraische Typen mit nur einem Konstruktor als Summentypen gelten.

3.2 Erweiterbare Union-Types

3.2.1 Geschlossene Summentypen

Es gibt bereits einige Versuche, algebraische Typen erweiterbar zu gestalten. Die bisherigen Ansätze lassen sich alle unter der Bezeichnung *erweiterbare Union-Types* zusammenfassen. Sie modellieren die Erweiterbarkeit mengentheoretisch. In der folgenden Typdefinition wird der algebraische Typ A aus 3.1 um einen neuen Konstruktor B_1 erweitert. $\oplus$ bezeichnet dabei die Typerweiterungsoperation.

```
data B = A ⊕ B₁ T₃,₁ ... T₃,r₃
```

Der neue algebraische Typ B besitzt drei Konstruktoren $\mathtt{A}_1$, $\mathtt{A}_2$ und $\mathtt{B}_1$. B entspricht damit dem Summentyp $B = A + B_1 = A_1 + A_2 + B_1$. In welcher Beziehung stehen nun die beiden Typen A und B zueinander?

Im folgenden bezeichnet $\leq$ die Untertyprelation. $A \leq B$ gilt, falls A ein Untertyp von B ist. Die Untertypbeziehung beschreibt die intuitive Vorstellung vom Enthaltensein eines Typs in einem anderen, wenn man sich Typen als Wertemengen vorstellt [CW85, Car97]. Rein mengentheoretisch muß also folgende Beziehung gelten:

$$A = A_1 + A_2 \quad \leq \quad A_1 + A_2 + B_1 = B$$

Folglich ist der erweiterte Typ B ein Obertyp von A. Abbildung 3.1 veranschaulicht die resultierenden Untertypbeziehungen. Algebraische Typen werden hierin als eckige Kästen dargestellt, die Typen der einzelnen Varianten werden als abgerundete Kästen abgebildet.

Der Umstand, daß der erweiterte Typ ein Obertyp des ursprünglichen algebraischen Datentyps darstellt, ist fatal. Das Beispiel aus Kapitel 2 verdeutlicht dies eindrucksvoll. Die folgenden beiden Typdeklarationen formulieren die Typen zur Darstellung der abstrakten Syntax für dieses Beispiel nochmals in der Haskell-Notation:

```
data Tree = Number int | Variable String | Plus Tree Tree | Times Tree Tree
data ExtendedTree = Tree ⊕ Lambda Tree Tree | Apply Tree Tree
```

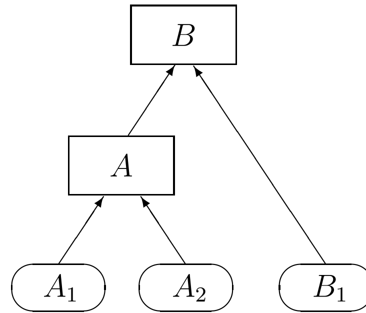


Abbildung 3.1: Untertypbeziehung für erweiterbare Union-Types

ExtendedTree ist gemäß obiger Feststellung ein Obertyp von *Tree*. Damit ist es nicht möglich, neue Konstrukte in alten zu schachteln. Beispielsweise ist es unmöglich, eine λ-Applikation in einem Plus-Konstrukt unterzubringen, da dieses *Tree*-Werte als Komponenten verlangt und keine Werte eines Obertyps akzeptiert. Der Grund für diesen Mißstand liegt schlichtweg darin, daß die Rekursion in Datentypen bereits bei der Definition geschlossen wird.

Ein ähnliches Problem ist auch auf der Seite der Funktionen zu finden. Keine existierende Funktion kann auf Werte des neuen erweiterten Typs angewendet werden. Eine Wiederverwendung von Funktionen ist ausgeschlossen.

Erweiterbare Union-Types erlauben es zwar, einen bestehenden algebraischen Typ zu erweitern, die Typerweiterung dürfte aber für die Praxis meist völlig unbrauchbar sein. Zum Schreiben von erweiterbaren Interpretern bzw. Übersetzern sind diese Art von Typen auf jeden Fall nicht einsetzbar.

3.2.2 Bisherige Ansätze

In der Literatur findet man Konzepte für erweiterbare algebraische Typen vor allem im Zusammenhang mit dem Bau modularer Interpreter in funktionalen Sprachen. Findler benötigt erweiterbare Datentypen, um in ML abstrakte Interpreter zu modularisieren [Fin95]. Er erweitert das Typsystem von SML [MTH90] so, daß zu jeder Zeit neue Konstruktoren zu einem bestehenden algebraischen Typ hinzugefügt werden können. Das Verfahren verändert einen bereits existierenden Typ und liefert keinen neuen Typ für die Erweiterung. Liang, Hudak und Jones beschreiben erweiterbare Union-Types, die sie mittels herkömmlicher algebraischer Typen modellieren [LHJ92]. Beide Ansätze erlauben es, neue Varianten für einen algebraischen Typ zu definieren, aber der endgültige Datentyp wird vor der Verwendung erst stets „geschlossen“. Die Lösungen leiden also an genau der oben beschriebenen Problematik.

Interessant sind in diesem Zusammenhang auch die *Mixin Modules* von Duggan und Sourelis [DS96] für SML [MTH90]. Dieser Ansatz erlaubt es, SML-Module in kleinere Teile, genannt Mixin Modules, zu zerlegen. Mixins ermöglichen es, rekursive Funktions- und

Typdefinitionen über einzelne modulare Fragmente zu verteilen. Mit Hilfe eines Kombinationsoperators lassen sich einzelne Mixins zusammensetzen. Ein Abschlußoperator erzeugt schließlich ein gewöhnliches SML-Modul aus einer Kombination verschiedener Mixins. Innerhalb eines Mixins sind algebraische Typen offen für mögliche Erweiterungen durch andere Mixins. Ebenso werden Funktionen durch die Einführung einer Pseudo-Variable `inner`[1] erweiterbar, über welche die entsprechende Funktion eines anderen Mixins aufgerufen werden kann. In [DS96] wird ein Beispiel für einen modular aufgebauten Interpreter gegeben. Dieses Beispiel ist fast identisch zur Interpreter-Implementation mit erweiterbaren algebraischen Typen aus Abschnitt 2.4. Die Funktion `eval` wird auf die gleiche Weise erweitert: `inner` spielt die gleiche Rolle wie `super.eval`.[2] Allerdings entsteht ein SML-Modul erst durch den Abschluß kombinierter Mixins. Dieses ist als solches nicht mehr erweiterbar. Auch die Menge der anwendbaren Funktionen ist auf die innerhalb der Mixins spezifizierten Funktionen beschränkt und nicht von außen um neue Funktionen ergänzbar.

3.3 Erweiterbare Algebraische Typen

3.3.1 Offene Summentypen

Ein algebraischer Typ wird klassischerweise durch eine feste Menge von Konstruktoren charakterisiert. Erweitert man einen Typ gemäß 3.2, erhält man wiederum einen algebraischen Typ, der durch eine feste Konstruktormenge gekennzeichnet ist. Wie der vorige Abschnitt gezeigt hat, sind diese erweiterten Typen in der Praxis kaum zu gebrauchen. Dies läßt sich jedoch ändern, wenn man mit erweiterbaren algebraischen Datentypen eine andere Vorstellung verbindet: nicht eine feste Konstruktormenge beschreibt einen Typ, sondern eine Mindestmenge an Konstruktoren. Erweitert man einen algebraischen Typ, so muß dieser mindestens die Menge an Konstruktoren unterstützen, die der ursprüngliche Typ definiert. Zusätzlich kann diese Konstruktormenge um neue Konstruktoren erweitert werden. Typtheoretisch bedeutet dies, daß algebraische Typen durch offene Typsummen zu modellieren sind. Im allgemeinen läßt sich ein erweiterbarer algebraischer Typ Y durch folgende Typsumme charakterisieren:

$$
\begin{aligned}
Y \;=\; & \mathit{inherited}_Y + \mathit{cases}_Y + \mathit{default}_Y \\
& \text{wobei} \quad \mathit{cases}_Y = \sum_i Y_i \\
& \qquad \mathit{inherited}_Y = \sum_{Y \preceq X} \mathit{cases}_X \\
& \qquad \mathit{default}_Y = \sum_{Z \preceq Y, Z \neq Y} \mathit{cases}_Z
\end{aligned}
$$

[1]Die Namensgebung geschieht wohl in Anlehnung an die Programmiersprache Beta. Dort referenziert man eine mögliche Erweiterung einer Funktion in einem Untermuster durch das `inner`-Konstrukt [BC90].

[2]Flatt, Krishnamurthi und Felleisen beschreiben in [FKF98] eine spezielle Form von Mixins für Java. Bei diesen Mixins wird tatsächlich die erweiterte Funktion mit dem `super`-Konstrukt aufgerufen, wie das bei erweiterbaren algebraischen Typen der Fall ist.

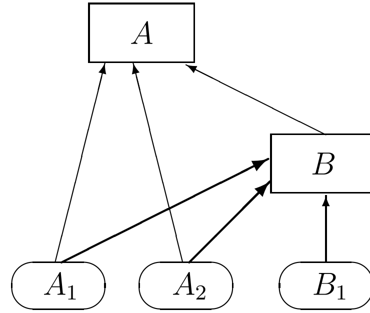


Abbildung 3.2: Untertypbeziehung für erweiterbare algebraische Typen

In dieser Definition erweiterbarer algebraischer Typen bezeichnet $\preceq$ die algebraische Erweiterungsrelation. Für zwei Typen X und Y gilt $Y \preceq X$ genau dann, wenn X und Y algebraische Datentypen sind und Y eine Erweiterung von X ist. Die Relation $\preceq$ ist also durch die Deklarationen der algebraischen Typen gegeben. Ein erweiterbarer algebraischer Typ Y setzt sich gemäß obiger Definition aus drei disjunkten Summentypen cases_Y, $\mathit{inherited}_Y$ und $\mathit{default}_Y$ zusammen. $\mathit{inherited}_Y$ bezeichnet die geerbten, cases_Y die zusätzlich definierten Konstruktoren. $\mathit{default}_Y$ ist dafür verantwortlich, daß die Typsumme insgesamt nicht geschlossen ist, indem es die Konstruktoren aller möglichen Erweiterungen von Y zusammenfaßt.

Für die Typen A und B des Beispiels aus Abschnitt 3.1 und 3.2 gilt damit:

$$\begin{array}{lcl} A & = & A_1 + A_2 + \mathit{default}_A \\ B & = & A_1 + A_2 + B_1 + \mathit{default}_B. \end{array}$$

Da die offene Typsumme $\mathit{default}_A$ sowohl B_1 als auch alle Typen aus $\mathit{default}_B$ enthält, gilt $B_1 + \mathit{default}_B \leq \mathit{default}_A$. Es gilt hier nicht $B_1 + \mathit{default}_B = \mathit{default}_A$, da $\mathit{default}_B$ gemäß obiger Definition nur Konstruktoren von Erweiterungen des Typs B zusammenfaßt. Konstruktoren anderer Erweiterungen von A werden durch $\mathit{default}_B$ nicht abgedeckt. Betrachtet man beispielsweise die Typ-Deklaration $\mathtt{data\ C\ =\ A} \oplus \mathtt{C_1}\ T_{4,1} \ldots T_{4,r_4}$ mit dem zugehörigen Typ $C = A_1 + A_2 + C_1 + \mathit{default}_C$, so erkennt man, daß C_1 nicht in $\mathit{default}_B$, wohl aber in der Typsumme $\mathit{default}_A$ enthalten ist. Damit ist $B_1 + \mathit{default}_B$ ein echter Untertyp von $\mathit{default}_A$, womit direkt folgt, daß $B \leq A$.

Verglichen mit erweiterbaren Union-Types ist die Untertypbeziehung zwischen algebraischem Basistyp A und einer Erweiterung B nun gerade umgedreht. Der erweiterte algebraische Typ ist ein Untertyp des ursprünglichen Datentyps. Abbildung 3.2 veranschaulicht die jetzt vorliegenden Untertypbeziehungen.

Interessant ist die Tatsache, daß nun die Typen der Varianten eines algebraischen Typs mehrere Obertypen besitzen. Diese kommen dynamisch durch eine Erweiterung des algebraischen Typs hinzu. Wird eine Variante Y_i in Typ Y definiert, so entspricht die Menge der direkten Obertypen von Y_i allen algebraischen Typen die Y erweitern.

$$\mathit{super}(Y_i) \;=\; \{X \mid X \preceq Y\}$$

Diese Menge ist im allgemeinen nicht endlich. Wie man in Abschnitt 3.4.2 sehen wird, ist es allerdings nicht erforderlich, daß man diese Menge zur Implementierung erweiterbarer algebraischer Typen statisch vollständig kennen muß.

Dadurch daß Erweiterungen jetzt Untertypen sind, hat die vorgestellte Lösung die in 3.2 für erweiterbare Union-Types beschriebenen Probleme nicht mehr. Der nachfolgende Abschnitt wird zeigen, daß die Rekursion in Datentypen jetzt offen für Erweiterungen ist. Außerdem können nun auch existierende Funktionen auf Werte des erweiterten Typs angewendet werden. Neue Varianten fallen beim Pattern Matching hier einfach in den Standardfall.

3.3.2 Konsequenzen

Exemplarisch sollen nun einige Beispiele diskutiert werden, die wichtige Eigenschaften erweiterbarer algebraischer Typen aufzeigen. Hierzu wird wiederum auf das durchgehende Beispiel aus Kapitel 2 zurückgegriffen. Folgende Definitionen liegen vor:

```
data Tree = Number int | Variable String | Plus Tree Tree | Times Tree Tree
data ExtendedTree = Tree ⊕ Lambda Tree Tree | Apply Tree Tree

toInt :: Tree -> Int
toInt Number i = i
toInt _ = error "number expected"

eval :: Tree -> Tree
...
```

Da der Typ *ExtendedTree* ein Untertyp von *Tree* ist, können im Gegensatz zu den Union-Types nun erweiterte Konstrukte in bisherigen Konstrukten geschachtelt werden. Folgender Ausdruck ist demnach zulässig:

```
id = Lambda (Variable "x") (Variable "x")
tree = Plus (Apply id (Number 7)) (Number 3)
```

Aus dem gleichen Grund können auch bestehende Funktionen auf Varianten des erweiterten algebraischen Typs angewendet werden. Die folgende Funktionsanwendung wird den Fehler „`number expected`“ liefern.

```
tree' = toInt (Lambda (Variable "x")
                      (Plus (Variable "x") (Number 1)))
```

Die hier vorgestellte Lösung besitzt also die in 3.2 für erweiterbare Union-Types aufgeführten Nachteile nicht mehr. Die Rekursion in Datentypen ist offen für neue Erweiterungen. Auch existierende Funktionen sind auf Werte des erweiterten Typs ohne weiteres anwendbar. Neue Varianten fallen beim Pattern Matching in den Standardfall.

Nun soll der algebraische Typ *Tree* unabhängig von *ExtendedTree* ein zweites Mal erweitert werden. Diesmal soll lediglich ein Subtraktions-Operator hinzukommen:

```
data ExtendedTree' = Tree ⊕ Minus ExtendedTree' ExtendedTree'
```

Der Typ *ExtendedTree′* erbt alle Konstruktoren von *Tree* und definiert zusätzlich noch einen `Minus`-Konstruktor. Obwohl sowohl der Typ *ExtendedTree* als auch der Typ *ExtendedTree′* von *Tree* abgeleitet wurden und eine ganze Reihe gemeinsamer Konstruktoren besitzen, sind sie nicht zueinander kompatibel. Folgender Ausdruck wäre illegal:

```
tree'' = Minus (Number 1) (Apply id (Number 2))
```

Unterschiedliche und damit voneinander unabhängige Erweiterungen eines algebraischen Typs werden also separiert. Es ist nicht möglich, Varianten unterschiedlicher Erweiterungen zu mischen. Sicherlich stellt dies eine Einschränkung dar.[3] Man muß jedoch bei jeder Erweiterung auch stets die Seite der Funktionen betrachten: Hat man eine Funktion `eval` für beide Typerweiterungen definiert, ergibt sich eben nicht automatisch gemäß dem Prinzip von Abschnitt 2.4.2 eine Operation `eval` für die Vereinigung der beiden Typerweiterungen. Man hat die gleichen Probleme wie bei erweiterbaren Union-Types. Im Sinne einer saubereren Lösung der Problematik macht es demnach also durchaus Sinn, unterschiedliche Erweiterungen voneinander zu trennen. Für die Art der Aufgabenstellung wie sie in Kapitel 2 formuliert wurde, ergibt sich damit sowieso keine Einschränkung.

3.4 Erweiterbare Algebraische Typen für Java

Bereits die Sprache Pizza erweitert Java um algebraische Datentypen [OW97]. Algebraische Typen werden hier durch Klassen definiert, die `case`-Konstrukte zur Deklaration der einzelnen Konstruktoren enthalten. Pattern Matching wird durch eine Erweiterung der `switch`-Anweisung zur Verfügung gestellt. Algebraische Typen sind in Pizza nicht erweiterbar.

In diesem Kapitel wird vorgestellt, wie erweiterbare algebraische Typen in die Programmiersprache Java im Stil von Pizza integriert werden können. In Pizza werden algebraische Typen durch eine relativ einfache Transformation in reguläres Java beschrieben. Da die Übersetzung erweiterbarer algebraischer Typen wesentlich komplizierter und nicht nur auf die Transformation der Typdeklarationen beschränkt ist, werden erweiterbare algebraische Typen für Java in diesem Abschnitt unabhängig von der Übersetzung spezifiziert. Die Kenntnis des Transformationsschemas von Abschnitt 3.5 ist keine Voraussetzung für das Verständnis der Typen.

[3] Diese Einschränkung ist vergleichbar mit den Restriktionen bei objektorientierten Sprachen die nur Einfachvererbung unterstützen. Zwei unterschiedliche Erweiterungen einer Klasse lassen sich nicht wieder vereinigen. Auch hier wird durch das Verbieten der Mehrfachvererbung vielen Problemen aus dem Weg gegangen.

3.4.1 Syntax und Semantik

Die in diesem Kapitel angegebenen Grammatiken erweitern die Java-Grammatik in [GJS96]. Dort werden alle übrigen Nicht-Terminale definiert, die in diesem Kapitel nicht weiter spezifiziert werden.

3.4.1.1 Algebraische Klassen

Ein erweiterbarer algebraischer Typ wird durch eine *algebraische Klasse* definiert. Eine Klasse ist genau dann algebraisch, wenn ihre Definition mindestens eine `case`-Deklaration enthält oder eine bestehende algebraische Klasse erweitert. Mittels des `case`-Konstrukts werden die Konstruktoren eines algebraischen Typs vereinbart. Ein erweiterbarer algebraischer Typ A mit zwei Varianten A_1 und A_2 läßt sich auf die folgende Art und Weise deklarieren:

```
class A {
   case A₁(f̄₁);
   case A₂(f̄₂);
}
```

Jede Variante $\mathtt{A_i}$ definiert einen *Falltyp* $A.A_i$ mit den Feldern $\bar{f}_i = T_{i,1}\, v_{i,1},\, \ldots\, , T_{i,r_i}\, v_{i,r_i}$, wobei die $T_{i,j}$ Typen und die $v_{i,j}$ Variablenbezeichner darstellen. Außerdem wird implizit ein Konstruktor $\mathtt{A.A_i}$ des Typs $T_{i,1} \times \ldots \times T_{i,r_i} \rightarrow A_i$ vereinbart, der ein neues Objekt vom Typ $A.A_i$ erzeugt. In der Notation von [GJS96] sieht eine Grammatik für allgemeine `case`-Deklarationen folgendermaßen aus:

$$\begin{array}{l} \mathit{CaseDeclaration}: \\ \quad \mathit{ClassModifiers}_{opt}\ \texttt{case}\ \mathit{Identifier}\ \mathit{CaseFormals}_{opt}\ \mathit{MethodBody} \end{array}$$

$$\begin{array}{l} \mathit{CaseFormals}: \\ \quad (\ \mathit{FormalParameterList}_{opt}\ ) \end{array}$$

Gemäß dieser Regel ist es möglich, in einem `case`-Konstrukt einen Methodenrumpf zu vereinbaren. Die Parameterliste des `case`-Konstrukts liegt im Gültigkeitsbereich des Methodenrumpfs. Wird mit einem Konstruktor ein neues Objekt erzeugt, so wird der zu dieser Variante gehörige Rumpf automatisch auf das neue Objekt angewendet. Über diesen Mechanismus ist es möglich, spezifische Initialisierungen vorzunehmen.

Die obige Grammatikregel läßt auch Konstruktoren ohne eine formelle Parameterliste zu. Die Variante X_1 im folgenden Beispielprogramm ist ein solcher Fall.

```
class X extends Object {
   case X₁;
   case X₂();
}
```

X_1 definiert keinen Falltyp, sondern stellt eine Konstante $\texttt{X.X}_1$ vom Typ X dar, die den Fall repräsentiert. X_2 definiert zwar auch keine Parameter, wird aber wie oben behandelt.

Als Modifikatoren sind für algebraische Klassen `public`, `abstract` und `final` zulässig. Eine algebraische Klasse muß genau dann `abstract` deklariert werden, wenn keine eigenen Varianten vereinbart werden. Dies ist nur für algebraische Klassen möglich, die eine andere algebraische Klasse erweitern. Der Modifikator `final` verbietet es, daß eine algebraische Klasse erweitert wird. Algebraische Datentypen, die `final` deklariert werden, entsprechen von der Semantik her den algebraischen Typen von Pizza. Formell wird ihr Typ durch eine geschlossene Typsumme beschrieben.

3.4.1.2 Erweiterung algebraischer Klassen

Die Typen A und X des vorangegangenen Abschnitts werden als *algebraische Basistypen* bezeichnet. Diese Typen werden durch algebraische Klassen definiert, deren Oberklassen nicht algebraischer Natur sind. Es ist also möglich, von jeder nicht algebraischen Klasse eine algebraische Basisklasse abzuleiten und damit daraus einen algebraischen Typ zu machen. Bildet man eine Unterklasse einer algebraischen Klasse, so ist diese per Definition wiederum eine algebraische Klasse. In dieser können neue Varianten definiert werden, um welche die Oberklasse erweitert wird. Die folgende Klassendefinition vereinbart einen *erweiterten algebraischen Typ B*, der von A alle Varianten erbt und zusätzlich noch eine neue Variante $\texttt{B}_1$ hinzufügt.

```
class B extends A {
   case B₁(f̄₃);
}
```

Es ist auch möglich, einzelne Varianten eines algebraischen Typs zu erweitern.[4] Im folgenden Beispiel wird eine Unterklasse A'_1 von A_1 vereinbart. A_1 ist eine Variante des algebraischen Typs A. A'_1 erweitert A_1 im Beispiel um einige zusätzliche Felder $\bar{f}'_1$.

```
class A'₁ extends A₁ {
   f̄'₁
   A'₁(f̄₁, f̄'₁) {
      super(v̄₁);
      ...
   }
}
```

Wie im folgenden Abschnitt beschrieben wird, ist diese Form der Erweiterung nicht nur zum Hinzufügen von weiteren Feldern nützlich, sondern kann auch zum spezifischen Überschreiben von Methoden eingesetzt werden. In diesem Fall wird eine Methode nur für eine bestimmte Variante spezialisiert.

[4] In der Terminologie von [DS96] wird diese Spezialisierung einer bestimmten Variante als *horizontale* Erweiterung eines algebraischen Typs bezeichnet, wohingegen das Hinzufügen neuer Varianten als *vertikale* Erweiterung gilt.

3.4.1.3 Attribute einer algebraischen Klasse

Algebraische Klassen können wie normale Java Klassen neben `case`-Deklarationen auch Variablen, Methoden und innere Klassen als Attribute besitzen.

ClassMemberDeclaration :
FieldDeclaration
MethodDeclaration
InnerClassDeclaration
CaseDeclaration

Instanzvariablen entsprechen Feldern, die jede deklarierte Variante zusätzlich zu den eigenen definierten Feldern besitzt. Da mit jeder Variante automatisch auch ein Konstruktor in Form einer statischen Methode erzeugt wird, sind gewöhnliche Java-Konstruktoren für algebraische Klassen unnötig. Ansonsten gibt es aber, verglichen mit regulären Java-Klassen, keine Restriktionen für *algebraische Basisklassen*. Für *erweiterte algebraische Klassen* ergeben sich allerdings einige weitere Einschränkungen:

- Neue Interfaces dürfen nur dann implementiert werden, wenn dadurch keine neuen Obertypen eingeführt werden.
- Es dürfen keine weiteren nicht-statischen Variablen vereinbart werden.
- Nicht-statische Methoden dürfen nur dann vereinbart werden, wenn sie Methoden einer Oberklasse überschreiben.

Diese Einschränkungen zielen darauf ab, die nicht-statische Schnittstelle von algebraischen Basisklassen und erweiterten algebraischen Klassen identisch zu halten. Dies ist notwendig, da Varianten einer algebraischen Klasse an algebraische Erweiterungen weitervererbt werden, sozusagen also zu mehreren algebraischen Klassen gehören. Da mit jedem Typ in Java, also auch Falltypen, eine eindeutige Schnittstelle assoziiert ist, bedeutet das, daß eine algebraische Basisklasse und alle ihre Erweiterungen die gleiche Schnittstelle besitzen müssen. Die Einschränkungen sind also eine logische Konsequenz von algebraischen Typen. Allerdings könnte man auf die beiden letzten Restriktionen der obigen Liste sicher verzichten, würde man nicht-statische Variablen und Methoden für algebraische Klassen insgesamt verbieten. Dies entspräche algebraischen Typen in funktionalen Sprachen. Allerdings wurde im Sinne einer besseren Integration in das objektorientierte Umfeld und der Kompatibilität mit Pizza wegen auf diese doch recht restriktiven Maßnahmen verzichtet.

Im folgenden muß noch das Überschreiben von Methoden in Zusammenhang mit algebraischen Klassen näher erläutert werden. Ausgehend von folgendem Beispielprogramm soll die Bedeutung überschriebener Methoden in erweiterten algebraischen Klassen motiviert werden.

```
class Alpha {
   case Case_1();
   void foo() {
      System.out.println("Alpha");
   }
}
class Beta extends Alpha {
   case Case_2();
   void foo() {
      System.out.println("Beta");
   }
}
```

Nun soll die folgende Anweisungsfolge betrachtet werden:

```
Beta b_2 = Beta.Case_2();
Beta b_1 = Beta.Case_1();
b_2.foo();
b_1.foo();
```

Der erste Aufruf von `foo` druckt wie erwartet „Beta“ aus. Man mag irrtümlicherweise erwarten, daß auch der zweite Aufruf das gleiche Resultat liefert. Dies ist jedoch nicht der Fall, da die Methode `foo` der Klasse `Beta` nur die Methode `Alpha.foo` für die in `Beta` definierten Varianten überschreibt. Für die Varianten aus `Alpha` wird weiterhin die in `Alpha` vereinbarte Methode aufgerufen. Daß dieses, auf den ersten Blick vielleicht überraschende Verhalten, für erweiterbare algebraische Typen genau so wie beschrieben ausfallen muß, verdeutlicht eine zweite Anweisungssequenz:

```
Beta  b = Alpha.Case_1();
Alpha a = b;
b.foo();
a.foo();
```

Man erwartet hier sicherlich, daß bei beiden Methodenaufrufen `foo` aus Klasse `Alpha` aufgerufen wird. Dies trifft auch zu, da beide Aufrufe bezüglich des gleichen Objekts erfolgen und der statische Typ beim Methodendispatch keine Rolle spielt. Die überladene Methode `foo` aus Klasse `Beta` wird damit niemals für Varianten der Klasse `Alpha` aufgerufen. Überschreiben hat in algebraischen Klassen also den Zweck, eine bestehende Methode für die neuen Varianten zu spezialisieren. Um eine Methode für eine bestehende Variante zu ändern, muß wie in 3.4.1.2 beschrieben, eine Unterklasse für die entsprechende Fallklasse angelegt werden, in der die Methode für diesen einen Fall überschrieben wird.

Auch für das Überladen von Methoden gibt es in Java mit erweiterbaren algebraischen Typen eine Sonderregelung. Diese ist jedoch prinzipieller Natur und nicht auf algebraische Klassen beschränkt. Auch dies soll anhand eines Beispielprogramms diskutiert werden.

```
class Test {
   void goo(Alpha a) {
      ...
   }
   void goo(Beta b) {
      ...
   }
   static void bar() {
      goo(Beta.Case2());
      goo(Beta.Case1());
   }
}
```

Die Klasse `Test` definiert zwei Instanzen für die überladene Methode `goo`. Für die beiden Aufrufe von `goo` in `bar` muß jeweils vom Übersetzer die statisch am besten passende Methode ermittelt werden. Für den Aufruf `goo(Beta.Case2())` paßt die Methode `goo(Beta b)` am besten, da *Beta* unmittelbarer Obertyp von $Case_2$ ist. Für den zweiten Aufruf `goo(Beta.Case1())` passen beide definierten Methoden allerdings gleich gut. $Case_1$ ist sowohl unmittelbarer Untertyp von *Alpha* als auch von *Beta* und es gibt kein offensichtliches Kriterium das bestimmt, welche Methode aufgerufen werden soll. Um dieses Problem zu vermeiden werden konkret Fälle dieser Art ausgeschlossen. Folgendes Kriterium beschreibt die Sonderregelung beim Überladen:

> Für jede Methode f mit überladenen Instanzen $f(T_1, \ldots, T_n)$ und $f(U_1, \ldots, U_n)$ muß mindestens für ein $i \in \{1, \ldots, n\}$ gelten, daß T_i und U_i inkompatibel sind.
>
> Zwei Typen T_1 und T_2 sind *inkompatibel* zueinander, geschrieben $T_1 \# T_2$, wenn sie ungleich und keine Erweiterungen des gleichen algebraischen Typs sind; d.h. es gilt $T_1 \# T_2 \Leftrightarrow \forall T : T_1 \preceq T \Rightarrow T_2 \not\preceq T$.

Abschnitt 3.5.2 wird zeigen, daß es zudem noch ein technisches Problem beim Übersetzen solcher überladener Methoden gibt. Durch die obige Überladungsrestriktion entfällt automatisch auch eine komplizierte Sonderbehandlung solcher Fälle.

3.4.1.4 Pattern Matching

Pattern Matching ist eine der Stärken algebraischer Typen. Es ermöglicht auf effiziente und einfach zu lesende Art und Weise, Fallunterscheidungen über die Varianten eines algebraischen Typs durchzuführen. Pattern Matching ist in funktionalen Sprachen ein elementarer Mechanismus. In einer imperativen, objektorientierten Sprache wie Java ist es sinnvoller, im Sinne einer komplikationslosen Integration, Pattern Matching durch ein spezielles Sprachkonstrukt zur Verfügung zu stellen. In Pizza wird dies durch eine Erweiterung der `switch`-Anweisung gemacht. Da sich diese Lösung bewährt hat und sehr flexibel ist, bietet es sich an, einen vergleichbaren Mechanismus auch für erweiterbare algebraische Typen in Java zu integrieren.

Die `switch`-Anweisung führt abhängig von dem Wert eines Ausdrucks, dem sogenannten *Selektor*, eine von mehreren Anweisungsfolgen aus.

SwitchStatement :
  `switch` (*Expression*) *SwitchBlock*

SwitchBlock :
  { $SwitchBlockStatementGroups_{opt}$ $SwitchLabels_{opt}$ }

SwitchBlockStatementGroups :
  SwitchBlockStatementGroup
  SwitchBlockStatementGroups *SwitchBlockStatementGroup*

SwitchBlockStatementGroup :
  SwitchLabels *BlockStatements*

SwitchLabels :
  SwitchLabel
  SwitchLabels *SwitchLabel*

SwitchLabel :
  `case` *TopLevelPattern* :
  `default` :

Der Typ des Selektors muß entweder `char`, `byte`, `short`, `int` oder ein algebraischer Typ sein. Mit jeder Anweisungssequenz ist eine Menge von *Mustern* verbunden. Die Typen der Muster müssen zum statischen Selektortyp passen. Ist der Selektortyp ein Basistyp, so heißt das, daß die Mustertypen dem Selektortyp zuweisbar sind. Bei einem algebraischen Selektortypen muß es sich bei den Mustertypen um Varianten dieses algebraischen Typs handeln.

Beim Auswerten der `switch`-Anweisung wird zunächst der Selektor ausgewertet. Nun wird der Selektorwert nacheinander von oben nach unten mit allen Mustern verglichen. Für das erste Muster das *paßt* wird die damit assoziierte Anweisungssequenz ausgeführt. Die `switch`-Anweisung muß, mit Ausnahme des letzten Falls, stets durch einen `break`-Befehl in der Anweisungssequenz verlassen werden.

Die folgenden Grammatikregeln beschreiben allgemein, wie die Muster einer `switch`-Anweisung aufgebaut sind.

TopLevelPattern :
  ConstantExpression
  ConstructorPattern

ConstructorPattern :
  ConstructorName $PatternParameterList_{opt}$

PatternParameterList :
($Patterns_{opt}$)

Patterns :
Pattern
Patterns , *Pattern*

Pattern :
TopLevelPattern
FormalParameter
_

Nur *TopLevelPattern* sind als Muster unmittelbar in einem `case`-Ausdruck zulässig. Sie bilden eine Untermenge aller möglichen Musterausdrücke, die geschachtelt definiert werden dürfen. Wie man der Grammatik entnehmen kann, ist ein Muster p allgemein entweder

- ein leeres Muster `_`,
- ein formeller Parameter bzw. eine Variable,
- ein konstanter Ausdruck gemäß §15.27 von [GJS96], oder
- ein Konstruktormuster der Form $c(p_1, \ldots, p_n)$, wobei c ein n-stelliger Konstruktor ist und die $p_1, \ldots, p_n$ selbst wiederum Muster sind.

Die in einem Muster vereinbarten Variablen müssen alle unterschiedlich sein. Sind mit einer Anweisungssequenz mehrere Muster assoziiert, so dürfen in den Mustern keine Variablen definiert sein. Innerhalb einer `switch`-Anweisung können Muster sich überlappen. Fallunterscheidungen müssen nicht unbedingt vollständig sein.

Der Abgleich eines Wertes v mit einem Muster p basiert im allgemeinen auf folgenden Regeln:

Fall 1: v ist von einem beliebigen *Basistyp* oder vom Typ `String`. v paßt genau dann zu p, wenn es sich bei p entweder um das leere Muster bzw. eine Variable handelt, oder falls p ein konstanter Ausdruck ist, dessen Wert mit dem von v übereinstimmt.[5]

Fall 2: $v = c(v_1, \ldots, v_n)$ ist von einem *algebraischen Typ A*. v paßt genau dann zu p, falls entweder

[5]Pizza erlaubt in Mustern lediglich konstante Ausdrücke der Typen `char`, `byte`, `short` und `int`. Für die in diesem Kapitel beschriebenen Muster werden diesbezüglich keine Einschränkungen gemacht. Sogar konstante Zeichenketten sind als Muster zulässig.

1. $p = _$ oder p eine Variable vom Typ A ist,
2. p eine Variable vom Falltyp c,[6] oder
3. p die Form $c(p_1, \ldots, p_n)$ besitzt und für alle $i \in \{1, \ldots, n\}$ gilt: v_i paßt zu p_i.

Fall 3: v ist von einem *nicht-algebraischen Referenztyp*. v paßt genau dann auf das Muster p, falls $p = _$ oder p eine Variable ist.

Abbildung 3.1 zeigt ein kleines Beispielprogramm, das einige Details abschließend nochmals verdeutlichen soll. Es wird ein einfacher algebraischer Typ $BinTree$ zur Repräsentation eines Binärbaums definiert. Dieser wird durch die Klasse `IntTree` um Blätter mit Integer-Werten erweitert. In der Klasse `Functions` werden auf den beiden algebraischen Typen einige Operation deklariert. Obwohl $BinTree$ nur eine Variante $Branch$ besitzt, wäre die Fallunterscheidung der `reflect`-Methode ohne den `default`-Fall nicht vollständig. Es wird ein `default`-Fall benötigt, da es möglich ist, $BinTree$ zu erweitern und die `reflect`-Methode auch auf Erweiterungen wie z.B. Objekte des Typs $IntTree$ anzuwenden. Man beachte auch, daß es in dieser `switch`-Anweisung nicht möglich ist, ein Muster der Form `Leaf`$(\ldots)$ zu verwenden. Der statische Typ des Selektors beschränkt der Typsicherheit wegen die zulässigen Varianten. Varianten von Erweiterungen werden

[6] Dieses Muster hat die Rolle einer Typschranke. Nur Werte eines bestimmten Falltyps passen. In den gängigen Programmiersprachen mit Pattern Matching kann diese Form von Muster oftmals mit Hilfe eines Alias-Operators imitiert werden.

```
class BinTree {
   case Branch(BinTree l, BinTree r);
}
final class IntTree extends BinTree {
   case Leaf(int i);
}
class Functions {
   BinTree reflect(BinTree tree) {
      switch (tree) {
         case Branch(BinTree l, BinTree r):
            return BinTree.Branch(reflect(r), reflect(l));
         default:
            return tree;
      }
   }
   boolean hasZero(IntTree tree) {
      switch (tree) {
         case Branch(BinTree l, BinTree r):
            return hasZero(l) || hasZero(r);
         case Leaf(0):
            return true;
         case Leaf(_):
            return false;
      }
   }
}
```

Programm 3.1: Pattern Matching für erweiterbare algebraische Typen

durch den `default`-Fall abgedeckt. Die Fallunterscheidung in `hasZero` dagegen ist auch ohne `default`-Fall vollständig. Der Typ $IntTree$ ist nicht erweiterbar und die drei Fälle schließen alle möglichen Werte ein.

3.4.2 Typsystem

Erweiterbare algebraische Typen lassen sich in Java recht einfach in das Typsystem integrieren. Alleine die Untertypbeziehung ist um algebraische Typen zu erweitern.

In diesem Abschnitt wird eine Terminologie gemäß [OW97] verwendet. Δ bezeichnet die globale Klassenumgebung, die Einträge der Form $c : \mathsf{class}(\Gamma, C, \bar{I})$ enthält. Γ stellt hierbei jeweils die lokale Umgebung der Klasse dar, C entspricht der Oberklasse und die $\bar{I}$ bezeichnen die implementierten Interfaces. Es sei $\mathcal{A} \subseteq \Delta$ die Menge aller algebraischen Klassen. Die lokale Umgebung Γ einer algebraischen Klasse $a : \mathsf{class}(\Gamma, C, \bar{I}) \in \mathcal{A}$ kann unter anderem Einträge $c : \mathsf{case}(\bar{f})$ enthalten, die den einzelnen Varianten des algebraischen Typs entsprechen. Da diese Varianten durch Klassen repräsentiert werden, gibt es für jede dieser Fallklassen zusätzlich noch einen vollständigen Eintrag in Δ.

Die aktuelle Umgebung Δ erzeugt die Untertyprelation $\leq$ zwischen Referenztypen für Java mit erweiterbaren algebraischen Typen gemäß Abbildung 3.3. Die ersten fünf Regeln genügen zur Beschreibung dieser Relation für reguläres Java. Die (Case)-Regel definiert die zusätzlichen Untertypbeziehungen zwischen den Varianten eines algebraischen Typs und allen Erweiterungen dieses Typs. Dies ist der einzige Zusatz zum Typsystem der nötig wird, um erweiterbare algebraische Typen in Java zu integrieren. Man beachte, daß die (Case)-Regel nicht die Semantik regulären Java-Codes beeinflußt, da neue Untertypbeziehungen nur für algebraische Typen eingeführt werden, die in regulärem Java nicht existieren.

$$\text{(Top)} \quad X \leq \texttt{java.lang.Object}$$

$$\text{(Refl)} \quad X \leq X$$

$$\text{(Trans)} \quad \frac{X_1 \leq X_2 \quad X_2 \leq X_3}{X_1 \leq X_3}$$

$$\text{(Super)} \quad \frac{c : \mathsf{class}(\Gamma, C, \bar{I}) \in \Delta}{c \leq C}$$

$$\text{(Intf)} \quad \frac{c : \mathsf{class}(\Gamma, C, \bar{I}) \in \Delta \quad X \in \bar{I}}{c \leq X}$$

$$\text{(Case)} \quad \frac{a_1 : \mathsf{class}(\Gamma, C, \bar{I}) \in \Delta \quad c : \mathsf{case}(\bar{f}) \in \Gamma \quad a_2 \preceq a_1}{c \leq a_2}$$

Abbildung 3.3: Erweiterte Untertyprelation $\leq$ für Java

Zur Definition der (Case)-Regel wird die *algebraische Erweiterungsrelation* $\preceq$ verwendet, die bereits aus Abschnitt 3.3.1 bekannt ist. $a_1 \preceq a_2$ gilt für zwei Typen a_1 und a_2 genau dann, wenn a_1 eine Erweiterung des algebraischen Typs a_2 ist. Abbildung 3.4 gibt eine formale Definition dieser Relation.

$$\text{(Refl)} \qquad a \preceq a$$

$$\text{(Trans)} \qquad \frac{a_1 \preceq a_2 \quad a_2 \preceq a_3}{a_1 \preceq a_3}$$

$$\text{(Extends)} \qquad \frac{a : \mathsf{class}(\Gamma_1, C, \bar{I}_1) \in \mathcal{A} \quad C : \mathsf{class}(\Gamma_2, C_2, \bar{I}_2) \in \mathcal{A}}{a \preceq C}$$

Abbildung 3.4: Algebraische Erweiterungsrelation $\preceq$

Daß $\preceq$ nicht bloß eine Einschränkung von $\leq$ auf die Menge der algebraischen Klassen $\mathcal{A}$ ist, soll das folgende Beispiel belegen. Hierin wird ein algebraischer Typ D vereinbart, der ein Untertyp des algebraischen Typs A, aber keine algebraische Erweiterung von A ist; d.h. es gilt $D \leq A$, aber nicht $D \preceq A$.

```
class A {
   case A₁(f̄₁);
   case A₂(f̄₂);
}
class D extends A₁ {
   case D₁(f̄₅);
}
```

Die partielle Ordnung $\preceq$ induziert eine Äquivalenzrelation $\simeq \subseteq \mathcal{A} \times \mathcal{A}$ auf algebraischen Typen. Mit $[\cdot]_\simeq$ werden die zugehörigen Äquivalenzklassen bezeichnet.

$$\begin{aligned} a_1 \simeq a_2 \quad &:\Leftrightarrow \quad \exists a\colon\ a_1 \preceq a \wedge a_2 \preceq a \\ [a]_\simeq \quad &:= \quad \{c \,|\, c \simeq a\} \end{aligned}$$

Jede Äquivalenzklasse $[a]_\simeq$ besitzt ein größtes Element *base*.

$$base = max([a]_\simeq) \Leftrightarrow \forall c \in [a]_\simeq\colon\ c \preceq base$$

base muß dem Typ einer algebraischen Basisklasse entsprechen, sonst gäbe es einen Typ $base' \in [a]_\simeq$ mit $base \preceq base'$. Dies widerspricht jedoch der Definition von *base*, das größte Element der Äquivalenzklasse zu sein. Alle weiteren Mitglieder der Klasse sind algebraische Erweiterungen des Basistyps. Die Äquivalenzrelation $\simeq$ partitioniert also die Menge der algebraischen Typen $\mathcal{A}$ so, daß sich in einer Äquivalenzklasse alle algebraischen Typen befinden, die von einem gemeinsamen algebraischen Basistyp abgeleitet werden. Im nachfolgenden Abschnitt werden diese Äquivalenzklassen dazu herangezogen, Typangaben für erweiterbare algebraische Klassen in reguläres Java zu übersetzen. Alle Typen einer Äquivalenzklasse werden hierbei auf einen einzigen Java-Typ abgebildet.

3.5 Übersetzung erweiterbarer algebraischer Typen

Dieses Kapitel beschreibt, wie erweiterbare algebraische Typen für Java effizient in reguläres Java transformiert werden können. Neben den Deklarationen algebraischer Typen betrifft die Transformation auch Typangaben, Typoperatoren und statische Qualifikationen. Die Übersetzung von Pattern Matching-Ausdrücken wird erst im nächsten Kapitel ausführlich diskutiert.

3.5.1 Deklarationen

Zur Beschreibung der Übersetzung algebraischer Klassen wird wieder das Beispiel aus den vorangegangenen Abschnitten herangezogen. Diesmal werden allerdings der Vollständigkeit wegen zusätzlich noch eine Variable und eine Methode definiert.

```
class A {
   case A1(f̄1);
   case A2(f̄2);
   Tvar x;
   Tfun m(f̄fun) {
      ...
   }
}
class B extends A {
   case B1(f̄3);
   Tfun m(f̄fun) {
      ...
   }
}
```

Jede algebraische Klasse wird durch eine gewöhnliche Java Klasse mit gleichen Variablen und Methoden implementiert. Die Varianten werden in innere TopLevel-Klassen übersetzt, die von der äußeren algebraischen Klasse abgeleitet sind. Die Variablen einer solchen Fallklasse entsprechen den in der Variante deklarierten Feldern. Für jede Fallklasse wird ein Konstruktor generiert, der genau die Felder als Parameter akzeptiert und den Instanzvariablen der Klasse zuweist. Für jede Variante wird in der algebraischen Klasse zusätzlich noch eine statische Methode gleichen Namens angelegt, die den Konstruktor der Fallklasse aufruft und das neu erzeugte Objekt zurückgibt. Jede einzelne Variante erhält ein Tag zugewiesen, über welches sie eindeutig identifiziert werden kann. Die Tags werden von Null beginnend durchnumeriert und können für jedes Objekt über eine Variable `$tag` abgefragt werden. Die Vergabe der Tags ist unabhängig von der Deklarationsreihenfolge der Varianten.[7] Eine Umordnung der Varianten verletzt also nicht die Binärkompatibilität einer algebraischen Klasse. Bei algebraischen Erweiterungen werden

[7] Die Varianten werden vor der Übersetzung lexikographisch sortiert.

```
class A {
   public final int $tag;
   protected A(int $tag) {
      super();
      this.$tag = $tag;
   }
   static class A1 extends A {
      f̄1
      public A1(f̄1) {
        super(0);
        this.v1,1 = v1,1;
        ...
      }
   }
   static class A2 extends A {
      f̄2
      public A2(f̄2) {
         super(1);
         this.v2,1 = v2,1;
         ...
      }
   }
   static A1 A1(f̄1) {
      return new A1(v̄1);
   }
   static A2 A2(f̄2) {
      return new A2(v̄2);
   }
   Tvar x;
   Tfun m(f̄fun) {
      ...
   }
}

class B extends A {
   protected B(int $tag) {
      super($tag);
   }
   static class B1 extends B {
      f̄3
      public B1(f̄3) {
         super(2);
         this.v3,1 = v3,1;
         ...
      }
   }
   static B1 B1(f̄3) {
      return new B1(v̄3);
   }
   Tfun m(f̄fun) {
      ...
   }
}
```

Programm 3.2: Übersetzung algebraischer Klassen

die Tags beginnend ab dem letzten Tag der algebraischen Oberklasse vergeben. Wird eine algebraische Klasse mehrmals unmittelbar erweitert, erhalten Varianten verschiedener Erweiterungen die gleichen Tags. Das in 3.4.2 vorgestellt Typsystem stellt jedoch statisch sicher, daß es niemals zum Vermischen von Varianten unterschiedlicher Erweiterungen kommen kann. Bereits in Abschnitt 3.3.2 wurde diese Eigenschaft anhand eines Beispiels erläutert. Die mehrfache Vergabe eines Tags für unabhängige Erweiterungen ist damit also völlig unproblematisch. Abbildung 3.2 zeigt den vollständigen Code für die beiden transformierten Beispielklassen.

3.5.2 Typen

Gemäß dem Typsystem aus 3.4.2 ist es möglich, daß eine Variante mehrere direkte Obertypen besitzt. Bei jeder Erweiterung des algebraischen Typs kommt ein neuer direkter Obertyp hinzu. Diese Eigenschaft wird durch die Übersetzung in 3.5.1 nicht erfüllt. Lediglich der definierende algebraische Typ ist unmittelbarer Obertyp der Varianten. Selbst wenn es in Java Mehrfachvererbung geben würde, könnte man damit die geforderte Eigenschaft nicht implementieren. Neue Obertypen können nämlich jederzeit dynamisch

hinzukommen, auch nachdem die algebraische Klasse schon übersetzt wurde. Da das Problem also nicht durch eine geschickte Klassendeklaration gelöst werden kann, müssen Typangaben allgemein eben so übersetzt werden, daß es möglich ist, die geforderte Eigenschaft zu erfüllen. Zur Verdeutlichung der Problematik soll zunächst folgende Anweisungssequenz betrachtet werden:

```
A a = B.B1(e1);
B b = B.A1(e2);
```

Die erste Anweisung ist eine legale Java-Anweisung. Objekte des Typs $B.B_1$ können Variablen des Typ A zugewiesen werden, da $B.B_1$ gemäß der Übersetzung aus 3.5.1 ein Untertyp des algebraischen Typs A ist. Die zweite Anweisung ist laut Typsystem ebenso völlig korrekt: B erbt die Varianten von A, womit es möglich sein muß, daß man einer Variable vom Typ B eine Variante A_i zuweist. Übersetzt man algebraische Typdeklarationen wie zuvor erläutert, so enthält diese Anweisung jedoch für reguläres Java einen Typfehler. Die Untertypbeziehung von $A.A_1$ zu B entsteht dynamisch durch die Erweiterung von A und kann, wie beschrieben, nicht statisch durch eine besondere Deklarationsübersetzung algebraischer Typen abgedeckt werden. Die einzige Möglichkeit die zweite Anweisung in korrekten Java-Code zu transformieren besteht darin, den Typ B in der Variablendeklaration durch A zu ersetzen. Die in reguläres Java übersetzten Anweisungen sehen damit folgendermaßen aus:

```
A a = B.B1(e1);
A b = B.A1(e2);
```

Für die Java-Seite ist diese Transformation sicher korrekt, da ja wie bereits in Abschnitt 3.4.1.3 erläutert, A und B, was den nicht-statischen Klassenanteil angeht, die gleiche Schnittstelle besitzen. Diese Übersetzungsregel löst das Problem neuer dynamischer Obertypen für statische Typdeklarationen dadurch, daß alle erweiterten algebraischen Datentypen auf ihre entsprechenden algebraischen Basistypen abgebildet werden. Dieser ist Obertyp aller definierten Varianten und besitzt bereits die gleiche Schnittstelle wie alle Erweiterungen. Es können also statisch keine Typisierungsprobleme auftreten. Formell läßt sich diese Übersetzungsregel für Typangaben folgendermaßen ausdrücken:

$$\frac{a \in \mathcal{A}}{a \longmapsto max([a]_{\simeq})}$$

(nicht im Kontext eines Typoperators oder einer Qualifikation einer statischen Variable oder Methode)

Es wird die Terminologie aus 3.4.2 verwendet. $\longmapsto$ bezeichnet die Abbildung eines Ausdrucks auf regulären Java-Code. Im Kontext eines Typecasts, eines `instanceof`-Operators oder in der Qualifikation statischer Variablen und Methoden darf diese Regel nicht angewendet werden. Die folgenden Abschnitte beschreiben, ob und wie Typen in diesen Fällen übersetzt werden.

Wichtigste Konsequenz der Übersetzungsregel ist, daß Typen erweiterter algebraischer Klassen niemals in übersetztem Java-Code auftauchen. Erweiterte algebraische Typen werden zur Übersetzungszeit nur zum Überprüfen der Typ-Korrektheit und zur Laufzeit nur für die Typoperatoren benötigt.

3.5.3 Typoperatoren

Der vorangegangene Abschnitt beschäftigte sich damit, wie man algebraische Typangaben übersetzen muß, um das Problem mit mehrfachen, dynamischen Obertypen bei Varianten algebraischer Klassen statisch in den Griff zu bekommen. Typecasts und `instanceof`-Operatoren müssen jedoch die Untertypbeziehungen dynamisch reflektieren. Das folgende Beispiel zeigt, daß auch hier eine spezielle Übersetzung notwendig ist.

```
class C extends A {
   case C₁(f̄₄);
}
...
A a = B.B₁(e₁);
B b = (B)a;
C c = (C)a;
```

Neben B erweitert auch C die algebraische Basisklasse `A` direkt. Das in der ersten Anweisung erzeugte $B.B_1$-Objekt wird einer Variable vom Typ A zugewiesen und anschließend zuerst auf B und dann auf C coerziert. Der zweite Typecast sollte fehlschlagen, da es sich bei $B.B_1$ nicht um eine Variante von C handelt. Übersetzt man jedoch die Anweisungssequenz gemäß der Regel aus 3.5.2, so ergibt sich folgendes Programm, das ausgeführt werden kann, ohne daß eine `ClassCastException` ausgelöst wird.

```
A a = B.B₁(e₁);
A b = (A)a;
A d = (A)a;
```

Für Narrowing-Casts auf algebraische Typen, nicht Falltypen, ist es notwendig, die vollständigen Untertypbeziehungen zwischen Varianten und algebraischen Typen zu kennen. Man könnte diese jederzeit aus der Klassenhierarchie ablesen. Es wäre jedoch bei weitem zu ineffizient, würde man das bei jedem Cast zur Laufzeit erneut tun. Für Casts ist also eine geeignete Unterstützung in Form eines Laufzeitsystems angebracht.

Unter Verwendung von Tabellen die angeben, welche Typen auf einen algebraischen Typ coerziert werden dürfen, ist es möglich, Narrowing-Casts auf algebraische Typen effizient zu implementieren. Programm 3.3 zeigt das hierfür nötige Laufzeitsystem für die beiden Beispielklassen `A` und `B`.

Jede algebraische Klasse wird zur Laufzeit mit einer Identifikationsnummer `$subID` versehen. Die Nummern werden von den algebraischen Basisklassen verwaltet. Jede algebraische Klasse, die eine algebraische Basisklasse erweitert, registriert sich nach dem Laden bei der Basisklasse und erhält daraufhin eine eindeutige Nummer zugewiesen. Für ein Objekt eines algebraischen Typs erhält man die Identifikationsnummer über die Methode `getSubID`. Zu jeder algebraischen Klasse gehört außerdem eine boolesche Tabelle `$castable`, die diejenigen algebraischen Klassen angibt, deren Varianten auf den eigenen Typ coerziert werden dürfen. Die Identifikationsnummern dienen als Zugriffscode auf

```
class A {
   public static A[] $extensions = {null, null, null, null};
   public static int $subIDs;
   public static boolean[] $castable = {true, false, false, false};
   public static final int $subID = $register(new A(-1));
   public static int $register(A x$0) {
      if ($subIDs == $extensions.length) {
         A[] new$arr = new A[$extensions.length * 2];
         System.arraycopy($extensions, 0, new$arr, 0, $extensions.length);
         $extensions = new$arr;
         for (int i = 0; i < $subIDs; i ++)
            $extensions[i].$extend();
      }
      $extensions[$subIDs] = x$0;
      return $subIDs++;
   }
   public static A throw$cast$exception() {
      throw new ClassCastException();
   }
   public void $extend() {
      boolean[] new$arr = new boolean[$castable.length * 2];
      System.arraycopy($castable, 0, new$arr, 0, $castable.length);
      $castable = new$arr;
   }
   public int $getSubID() {
      return $subID;
   }
   ...
}

class B extends A {
   public static boolean[] $castable;
   public static final int $subID = $register(new B(-1));
   static {
      $castable = new boolean[A.$castable.length];
      $castable[A.$subID] = $castable[B.$subID] = true;
      A.$castable[$subID] = true;
   }
   public void $extend() {
      boolean[] new$arr = new boolean[$castable.length * 2];
      System.arraycopy($castable, 0, new$arr, 0, $castable.length);
      $castable = new$arr;
   }
   public int $getSubID() {
      return $subID;
   }
   ...
}
```

Programm 3.3: Laufzeitsystem für erweiterbare algebraische Klassen

die Tabellen. Die Tabellen werden ebenfalls zur Laufzeit konstruiert. Beim Laden eines neuen algebraischen Typs trägt dieser sich nach der Registrierung bei der algebraischen Basisklasse automatisch in allen relevanten Tabellen ein. Da zur Übersetzungszeit die Größe der Tabellen nicht bekannt und im allgemeinen auch nicht beschränkt ist, müssen die Tabellen nötigenfalls dynamisch vergrößert werden. Dieser Vorgang wird zentral von der algebraischen Basisklasse für alle bekannten Erweiterungen bei einem Überlauf der eigenen Tabelle angestoßen. Auf diese Weise sind alle Tabellen stets groß genug und es kann niemals zu einem Überlauf kommen. Aufbauend auf diesem Laufzeitsystem läßt sich ein allgemeines Übersetzungsschema für Typecasts der Form $(a)x$ angeben. x stellt dabei einen beliebigen Ausdruck mit statischem Typ t_x dar.

$$\frac{a \preceq t_x \qquad a \neq t_x \qquad x \longmapsto y}{(a)x \;\longmapsto\; \begin{array}{l}\texttt{((\$temp = }y\texttt{) == null) || }a\texttt{.\$castable[\$temp.\$getSubID()] ?}\\ \quad\texttt{\$temp : }a\texttt{.throw\$cast\$exception()}\end{array}}$$

$$\frac{a \in \mathcal{A} \qquad t_x \notin \mathcal{A} \qquad a \leq t_x \qquad a \neq b = max([a]_{\simeq}) \qquad x \longmapsto y}{(a)x \;\longmapsto\; \begin{array}{l}\texttt{((\$temp = }y\texttt{) == null) || }a\texttt{.\$castable[((}b\texttt{)\$temp).\$getSubID()] ?}\\ \quad\texttt{(}b\texttt{)\$temp : }b\texttt{.throw\$cast\$exception()}\end{array}}$$

Die Hilfsvariable `$temp` muß jeweils zuvor an geeigneter Stelle mit einem passenden Typ vereinbart werden. Typtests lassen sich auf eine analoge Art und Weise transformieren. Das folgende Übersetzungsschema zeigt, wie man `instanceof`-Operatoren unter Verwendung des Laufzeitsystems effizient übersetzen kann:

$$\frac{a \preceq t_x \qquad a \neq t_x \qquad x \longmapsto y}{x\ \texttt{instanceof}\ a \;\longmapsto\; \texttt{((\$temp = }y\texttt{) != null) \&\& }a\texttt{.\$castable[\$temp.\$getSubID()]}}$$

$$\frac{a \in \mathcal{A} \qquad t_x \notin \mathcal{A} \qquad a \leq t_x \qquad a \neq b = max([a]_{\simeq}) \qquad x \longmapsto y}{x\ \texttt{instanceof}\ a \;\longmapsto\; \texttt{((\$temp = }y\texttt{) != null) \&\& }a\texttt{.\$castable[((}b\texttt{)\$temp).\$getSubID()]}}$$

Für Fälle in denen die Übersetzungsregeln nicht anwendbar sind, wird keine spezielle Transformation benötigt. Insbesondere darf keine Typtransformation bei `.class`-Ausdrücken erfolgen.

3.5.4 Statische Qualifikationen

Ausgenommen bei Typtransformationen sind Fälle, bei denen der Klassenname als Qualifikation für statische Felder oder Methoden herangezogen wird. Nach der Java Sprachspezifikation ist es jedoch möglich, statische Klassenkomponenten auch über einen beliebigen Ausdruck zu qualifizieren. In diesem Fall bestimmt der statische Typ des Ausdrucks, auf welche Klasse zugegriffen wird. Die Transformation für erweiterbare algebraische Typen erhält jedoch im allgemeinen nicht den statischen Typ eines Ausdrucks. Nach 3.5.2 kann in transformiertem Code niemals der Typ einer erweiterten algebraischen Klasse auftauchen. Will man auf eine statische Klassenkomponente einer erweiterten algebraischen

Klasse über einen qualifizierenden Ausdruck zugreifen, so scheitert die bisherige Übersetzung. Es wird die algebraische Basisklasse referenziert und nicht die gewünschte erweiterte algebraische Klasse. Dieser Spezialfall erfordert eine gesonderte Behandlung. Die Qualifikation der statischen Klassenkomponente muß explizit über den Klassennamen erfolgen. Der ursprünglich qualifizierende Ausdruck muß zuvor ausgewertet werden. In den beiden folgenden Regeln steht T_m^{res} für den Rückgabetyp der Methode m.

$$\frac{T_m^{res} = void \qquad t_{\mathtt{x}} \in \mathcal{A} \qquad t_{\mathtt{x}} \neq max([t_{\mathtt{x}}]_{\simeq}) \qquad \mathtt{x} \longmapsto \mathtt{y}}{x.m(\ldots) \longmapsto y;\ t_{\mathtt{x}}.m(\ldots)}$$

$$\frac{T_m^{res} \neq void \qquad t_{\mathtt{x}} \in \mathcal{A} \qquad t_{\mathtt{x}} \neq max([t_{\mathtt{x}}]_{\simeq}) \qquad \mathtt{x} \longmapsto \mathtt{y}}{x.m(\ldots) \longmapsto ((\mathtt{\$temp} = y) == \mathtt{\$temp})\ ?\ t_{\mathtt{x}}.m(\ldots)\ :\ null}$$

In der zweiten Regel steht *null* für einen Ausdruck der den Wert Null im Typ des Rückgabewertes der Methode m liefert. Dieser Ausdruck wird nur der Vollständigkeit wegen benötigt. Er wird niemals ausgewertet, da der boolesche Ausdruck des `?:`-Operators stets wahr ist. Überhaupt erscheint der übersetzte Code der zweiten Regel etwas ungewöhnlich. Es ist leider die einzige Möglichkeit, ohne zusätzliche Hilfsmittel aus einem Ausdruck zwei Ausdrücke zu machen, die nacheinander ausgewertet werden, wobei das erste Resultat verworfen und das Ergebnis des zweiten Ausdrucks zurückgegeben wird.

Statische Variablenzugriffe werden analog behandelt. Alle übrigen Konstrukte der Sprache Java bedürfen keiner speziellen Übersetzung. Ausnahme ist die `switch`-Anweisung, über welche Pattern Matching auf algebraischen Typen zur Verfügung gestellt wird.

3.6 Übersetzung von Pattern Matching

In Abschnitt 3.4.1.4 wurde Pattern Matching für erweiterbare algebraische Typen in Java spezifiziert. Dieses Kapitel beschäftigt sich damit, wie man Pattern Matching-Ausdrücke in Form von `switch`-Anweisungen in effizienten, regulären Java-Code überführen kann. In [Aug85] und [PJ86] wird beschrieben, wie sich Pattern Matching in funktionalen Sprachen in einfache `case`-Ausdrücke transformieren läßt. Diese lassen sich dann in einem zweiten Schritt weiter in das Lambda-Kalkül übersetzen. Field und Harrison beschreiben in [FH88] ein alternatives Verfahren, bei welchem die Transformation in `case`-Ausdrücke über eine Baumdarstellung erfolgt. Das angegebene Verfahren ist allerdings nur bei nichtüberlappenden Mustern bzw. *Best-Fit*[8]-Pattern Matching anwendbar.

Auch für Pattern Matching in Java soll ein zweistufiges Übersetzungsschema verwendet werden. Im ersten Schritt findet eine Übersetzung in ein Zwischenformat statt, welches bereits nur noch aus einfachen Fallunterscheidungen besteht. Im zweiten Schritt wird

[8]Beim Best-Fit-Pattern Matching ist bei überlappenden Mustern nicht die Reihenfolge der Muster ausschlaggebend, sondern es wird das Muster gewählt, das gemäß einem Kriterium am besten paßt.

aus diesem Format Java-Code erzeugt. Im Gegensatz zu den Lösungen in [Aug85] und [PJ86] wird die Zwischendarstellung jedoch inkrementell aufgebaut. Dies ermöglicht eine einfache und effiziente Implementierung der Transformation in einer imperativen Sprache wie Java. Auch das Verfahren in [FH88] baut die Zwischendarstellung inkrementell auf, allerdings werden hier die einzelnen Muster zunächst getrennt in eine Zwischendarstellung in Form eines Baumes überführt und anschließend die einzelnen Bäume zu einem Musterbaum verschmolzen.

Durch den Umweg über die Zwischenrepräsentation wird das Erzeugen der allgemeinen Struktur eines übersetzten Pattern Matching-Ausdrucks von der konkreten Codeerzeugung für Java entkoppelt. Diese Vorgehensweise hat mehrere Vorteile:

- Die Transformation ist durch die Dekomposition des Übersetzungsvorgangs einfacher zu verstehen.
- Der zu erzeugende Code kann einfacher optimiert werden.
- Mit dem Schema kann auch Code für eine andere Zielsprache generiert werden. Hierfür muß lediglich die zweite Phase ausgetauscht werden.

Anhand der Zwischenrepräsentation läßt sich zudem relativ einfach feststellen, ob eine Fallunterscheidung vollständig ist. Da dieser *Exhaustive Check* normalerweise sehr aufwendig ist, werden in vielen Übersetzern, wie beispielsweise im Pizza-Übersetzer, approximative Methoden verwendet. Der in 3.6.3 beschrieben Algorithmus liefert dagegen stets exakte Ergebnisse. Da der Algorithmus auf der Zwischendarstellung arbeitet, bekommt man das Ergebnis ohne großen zusätzlichen Aufwand. Mit dem Beispiel im nächsten Abschnitt wird unter anderem motiviert, daß man zum Übersetzen von Pattern Matching-Ausdrücken in korrekte Java-Programme im allgemeinen einen exakten Exhaustive Check benötigt.

3.6.1 Übersetzung von `switch`-Blöcken

Das Übersetzen von `switch`-Blöcken ist relativ kompliziert. Deswegen wird in 3.6.1.1 zunächst anhand eines Beispiels gezeigt, wie der übersetzte Code eines Pattern Matching-Ausdrucks typischerweise aussieht. Abschnitt 3.6.1.2 stellt die Struktur der Zwischenrepräsentation vor und erläutert, wie Zwischendarstellungen zu interpretieren sind. Anschließend wird in 3.6.1.3 anhand von verschiedenen Beispielen ein allgemeiner Transformationsalgorithmus abgeleitet, der `switch`-Blöcke in die Zwischendarstellung überführt.

3.6.1.1 Ein Beispiel

Bevor das Transformationsverfahren näher erläutert wird, soll zunächst anhand eines kleinen Beispiels veranschaulicht werden, welche grundlegenden Ideen hinter der Über-

setzung von `switch`-Anweisungen stecken. Das folgende Programm definiert eine algebraische Klasse `List` mit einer Methode `nodups`, welche Sequenzen von identischen Elementen in einer Liste auf ein Element reduziert.

```
final class List {
   case Nil;
   case Cons(Object head, List tail);

   List nodups() {
      switch (this) {
         case Nil:
            return Nil;
         case Cons(_, Nil):
            return this;
         case Cons(Object x, Cons(Object y, List ys)):
            return (x == y) ? Cons(y, ys).nodups()
                            : Cons(x, Cons(y, ys).nodups());
      }
   }
}
```

Gemäß den Beschreibungen von 3.4.1.4 muß für `nodups` Code erzeugt werden, der den Selektorausdruck `this` zunächst mit `Nil` vergleicht. Schlägt der Vergleich fehlt, muß `this` mit dem geschachtelten Muster `Cons(_, Nil)` abgeglichen werden. Ist auch dies nicht möglich, findet schließlich ein Vergleich mit dem letzten Muster statt. Würde man `switch`-Anweisungen auf diese Weise implementieren, wäre das zwar korrekt, aber sehr ineffizient. `this` würde beispielsweise sowohl beim zweiten als auch beim dritten Muster als erstes mit einem `Cons`-Element abgeglichen werden.

Die grundlegende Idee bei der Übersetzung von `switch`-Anweisungen besteht darin, Muster mit dem gleichen äußeren Konstruktor zusammenzufassen. Die Felder eines Konstruktors müssen anschließend mit den jeweiligen Untermustern abgeglichen werden. Die geschachtelten Abgleiche werden analog übersetzt. Von welchem Konstruktor ein Objekt erzeugt wurde, kann stets über das Tag-Feld des Objekts bestimmt werden. Das nachfolgende Programm zeigt die übersetzte `nodups`-Methode.

```
List nodups() {
   switch (this.$tag) {
      case 1:     /* Nil */
         return Nil;
      case 0: {   /* Cons */
         final List.Cons temp$0 = (List.Cons)this;
         switch (temp$0.tail.$tag) {
            case 1:     /* Nil */
               return this;
            case 0: {   /* Cons */
               final List.Cons temp$2 = (List.Cons)temp$0.tail;
```

```
                Object x = temp$0.head;
                Object y = temp$2.head;
                List ys = temp$2.tail;
                return (x == y) ? Cons(y, ys).nodups()
                                : Cons(x, Cons(y, ys).nodups());
            }
        }
        break;
    }
  }
  throw new Error();
}
```

Der Pattern Matching-Ausdruck wird durch eine reguläre `switch`-Anweisung über den Tag des äußersten Konstruktors implementiert. Innerhalb dieser Fallunterscheidung muß im allgemeinen für jeden vorliegenden Fall der Selektor als erstes auf den zugehörigen Variantentyp gecasted werden. Diese Casts werden im obigen Programm unterstrichen dargestellt. Anschließend werden die einzelnen Felder der Variante mit ihren entsprechenden Untermustern abgeglichen. Dies geschieht wiederum durch `switch`-Anweisungen über die Tags der Felder. Im obigen Beispiel genügt es, lediglich das Feld `temp$0.tail` zu betrachten. Für alle übrigen Felder des `Cons`-Objekts sind keine Abgleiche nötig.

Von dem Code, der den eigentlichen Musterabgleich implementiert, dem sogenannten *Matching Code* [FH88], sind die Anweisungen zu unterscheiden, die ausgeführt werden, wenn ein Muster erfolgreich abgeglichen wurde. Wie man gut am obigen Programm erkennen kann, besteht diese Anweisungssequenz jeweils aus zwei Teilen: dem *Binding Code*, der die im Muster enthaltenen Variablen deklariert und initialisiert und dem *Body Code*, der einem Rumpf der ursprünglichen Fallunterscheidung entspricht.

Wendet man das informell erläuterte Übersetzungsprinzip an, so entsteht nicht notwendigerweise ein legales Java-Programm. Würde man beispielsweise die kursiv gedruckte Anweisung in der Übersetzung weglassen, wäre das Programm nicht korrekt. Es wäre theoretisch möglich, die Methode `nodups` ohne eine `return`-Anweisung zu verlassen. Betrachtet man die ursprüngliche Fallunterscheidung, so wird man jedoch feststellen, daß dies gar nicht möglich ist, da es sich hier um eine vollständige Fallunterscheidung handelt. Diese Tatsache muß man im erzeugten regulären Java-Code explizit durch das Auslösen einer Exception ausdrücken, auch wenn diese Anweisung selbst niemals ausgeführt wird.

3.6.1.2 Zwischenrepräsentation

Bei der Zwischenrepräsentation handelt es sich um einen gerichteten Graphen, der aus fünf verschiedenen Knotentypen aufgebaut wird. Von jedem der Knoten gehen zwei Kanten zu Nachfolgern aus. Die Graphen besitzen damit eine Baumstruktur. Außerdem ist mit jedem Knoten ein Typ assoziiert. Die Zwischenrepräsentation läßt sich durch folgende Datenstruktur beschreiben:

```
class PatternNode {
   Type       type;
   PatternNode and;
   PatternNode or;

   case Switch(Tree selector, Name temp);
   case DefaultPat();
   case ConstrPat(int tag, int args);
   case ConstantPat(Constant value);
   case Body(VarDecl[] bindings, Tree[] stats);
}
```

Fallunterscheidungen über beliebige Typen werden in der Zwischendarstellung durch *Switch*-Knoten repräsentiert. Über den `or`-Nachfolger eines Switch-Knotens wird der erste Fall der Fallunterscheidung referenziert. Der `and`-Knoten verweist auf eine folgende Fallunterscheidung. Muster werden mit Hilfe der *DefaultPat*-, *ConstrPat*- und *ConstantPat*-Knoten beschrieben. DefaultPat-Knoten stehen für Variablen und leere Muster, ConstrPat-Knoten repräsentieren einfache Konstruktormuster und mit ConstantPat-Knoten lassen sich Konstanten in Musterausdrücken formulieren. Auf der Ebene der Zwischenrepräsentation gibt es bereits keine geschachtelten Muster mehr. Diese werden durch eine geeignete Verkettung der ...`Pat`-Knoten über die `and`- und `or`-Felder dargestellt. Der `and`-Verweis gibt stets ein weiteres Muster an, das abgeglichen werden muß, bevor das Gesamtmuster paßt. Der `or`-Verweis zeigt auf ein alternatives Muster. Die *Body*-Knoten stehen in der Zwischenrepräsentation für die Rümpfe, d.h. die Anweisungssequenzen der einzelnen Fälle. Zwischenrepräsentaionen werden in diesem Kapitel gemäß folgender Notation angegeben:

- `SWITCH`$(Typ, Selektor, Variable)$ beschreibt einen Switch-Knoten für eine Fallunterscheidung bezüglich des *Selektor*-Ausdrucks. *Typ* gibt den Datentyp an, zu welchem die einzelnen Fälle gehören müssen. *Variable* bezeichnet eine Hilfsvariable, auf die der *Selektor*-Ausdruck beim Zutreffen eines Falls gecasted wird (siehe Beispiel unten).

- Body-Knoten werden stets mit `BODY` bezeichnet. Die konkreten Anweisungssequenzen sind hier nicht von Interesse.

- Konstruktor-Knoten erscheinen in der Form $Konstruktor(Tag, \#Argumente)$.

- DefaultPat-Knoten werden mit _ bezeichnet.

- ConstantPat-Knoten werden durch die Repräsentation der Konstante selbst dargestellt.

- `and`-Verweise werden durch → zwischen zwei Knoten angedeutet. Die `or`-Verzeigerung wird nicht explizit angegeben. Alle Musterknoten die alternativ verzeigert sind werden auf einen Switch-Knoten folgend gleichermaßen eingerückt untereinander aufgelistet.

Für die switch-Anweisung der Methode nodups aus dem vorigen Abschnitt ergibt sich folgende Zwischenrepräsentation.[9]

```
SWITCH(List, this, temp$0)
  Nil(1, 0) → BODY
  Cons(0, 2)→ SWITCH(Object, temp$0.head, temp$1)
               _→ SWITCH(List, temp$0.tail, temp$2)
                    Nil(1, 0) → BODY
                    Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                 _→ SWITCH(List, temp$2.head, temp$4)
                                      _→ BODY
```

Der Graph dieser Darstellung läßt sich folgendermaßen interpretieren: Für den Selektor this muß zunächst getestet werden, ob es sich um ein Nil- oder ein Cons-Objekt handelt. Tritt der erste Fall ein, wurde bereits ein passendes Muster gefunden, da ein Body-Knoten erreicht wird. Im Falle von Cons muß als nächstes die Fallunterscheidung bezüglich des head-Feldes betrachtet werden. Hier gibt es nur ein leeres Muster zur Auswahl, welches auf jeden Fall paßt.[10] Über den and-Verweis erreicht man die nächste Fallunterscheidung, diesmal über die tail-Komponente des ersten Cons-Objekts. Liegt hierfür ein Nil-Wert vor, trifft man auf einen Body-Knoten, der für den Rumpf des damit abgeglichenen Musters steht. Im Cons-Fall müssen dagegen zwei weitere Fallunterscheidungen, die jedoch beide nur ein leeres Muster zur Auswahl haben, abgearbeitet werden, bevor man schließlich den Body-Knoten für den dritten Fall der ursprünglichen Fallunterscheidung erreicht.

Mit der Zwischenrepräsentation hat man also bereits einen Algorithmus, der auf einer relativ abstrakten Ebene beschreibt, wie man die ursprüngliche switch-Anweisung auszuwerten hat. Die Darstellung läßt sich unmittelbar in ein korrespondierendes Java-Programm überführen. Das Hauptproblem besteht in der Transformation eines Pattern Matching-Ausdrucks in die Zwischendarstellung. Dabei müssen geschachtelte Muster und Tupel von Mustern (bei Konstruktoren mit mehreren Feldern) in eine flache Struktur übersetzt werden. Abschnitt 3.6.1.3 gibt einen allgemeinen Übersetzungsalgorithmus an, der für switch-Anweisungen inkrementell – also Fall für Fall – die korrespondierende Zwischenrepräsentation aufbaut.

3.6.1.3 Überführung in die Zwischenrepräsentation

Eingabe für den Transformationsalgorithmus ist ein korrekt typisierter switch-Block der Form:

---

[9] Erst der nächste Abschnitt wird zeigen, wie man zu dieser Darstellung kommt.

[10] Man beachte, daß eine Variable Object head beim Musterabgleich wie ein leeres Muster behandelt wird.

```
switch (e) {
   case p1: A1;
   case p2: A2;
  ...
   case pn: An;
}
```

Die Idee des Algorithmus besteht darin, die Fälle in ihrer Definitionsreihenfolge nacheinander in den Graphen der Zwischenrepräsentation einzufügen. Begonnen wird mit dem Knoten `SWITCH(`t_e`, `e`, temp$0)`, wobei t_e hier der algebraische Typ des Selektorausdrucks e ist. Eingefügt werden die Fälle mit einer überladenen, rekursiven Methode `enter` folgender Signatur:

```
PatternNode enter(Tree actual, Definition formal, PatternNode target,
                  Switch header, Env env);
PatternNode enter(Tree[] actuals, Definition[] formals, PatternNode target,
                  Switch header, Env env);
```

`enter` fügt das aktuelle Muster `actual`, welches zum formellen Parameter `formal` gehört, an der Stelle `target` in der Zwischenrepräsentation ein. `header` gibt bei einem geschachtelten Muster den Switch-Knoten des umgebenden Konstruktors an. Über den Parameter `env` werden für einen Fall der ursprünglichen `switch`-Anweisung Variablenbindungen gesammelt, die durch Muster eingeführt werden. Bevor der vollständige Algorithmus der Methode `enter` angegeben wird, werden anhand eines Beispiels alle möglichen Situationen ausführlich diskutiert. Abbildung 3.4 zeigt das vollständige Beispielprogramm. Die Methode `zip` der Klasse `List` akzeptiert als Parameter zwei Listen `xs` und `ys` und erzeugt eine Liste von Paaren der Elemente beider Listen. Es wird nun Schritt für Schritt die Zwischendarstellung für den Pattern Matching-Ausdruck in `zip` konstruiert.

```
final class ListPair {
   case Pair(List fst, List snd);        /* Tag 0 */
}
final class List {
   case Nil;                             /* Tag 1 */
   case Cons(Object head, List tail);    /* Tag 0 */

   List zip(List xs, List ys) {
      switch (ListPair.Pair(xs, ys)) {
         case Pair(Cons(_, _), Nil):
            return Nil;
         case Pair(Nil, _):
            return Nil;
         case Pair(Cons(Object h1, List t1), Cons(Object h2, List t2)):
            return Cons(new Object[]{h1, h2}, zip(t1, t2));
      }
   }
}
```

Programm 3.4: Beispiel für Pattern Matching in Java

Eintragung des ersten Falls

Begonnen wird die Übersetzung der `switch`-Anweisung, indem der Startknoten der Zwischenrepräsentation erzeugt wird. Ist der Selektorausdruck wie in Programm 3.4 kein Bezeichner, so wird der Selektor zunächst einer Hilfsvariable `temp$0` zugewiesen.

```
SWITCH(ListPair, temp$0, temp$1)
```

Unterstrichene Knoten geben die Stelle im Graphen an, an der der nächste Knoten angefügt werden muß. In der `enter`-Methode entspricht dies dem Parameter `target`. Im initialen Graphen werden nun nacheinander die einzelnen Fälle der `switch`-Anweisung eingetragen. Muster werden hierfür stets in ihre Prefixform zerlegt. Für den ersten Fall bedeutet das, daß zunächst der äußerste `Pair`-Konstruktor des geschachtelten Musters als erste Alternative an den Switch-Knoten angefügt wird.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)
```

Ausgehend von dem neuen unterstrichenen Zielknoten gilt es nun die beiden noch verbleibenden Teilmuster `Cons(_, _)` und `Nil` einzutragen. Da Muster von links nach rechts abgeglichen werden, muß zunächst ein Abgleich auf das `Cons`-Muster über den `and`-Verweis des aktuellen Zielknotens erreichbar sein. Da über den `and`-Verweis bisher noch nichts referenziert wird, wird hierfür ein passender Switch-Knoten erzeugt, der die geschachtelte Fallunterscheidung repräsentiert. Der Konstruktor-Knoten für `Cons` wird als erste Alternative eingetragen.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
               Cons(0, 2)
```

Nach dem gleichen Prinzip müssen nun noch die beiden leeren Muster des `Cons`-Musters bearbeitet werden:

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
               Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                  _→ SWITCH(List, temp$2.tail, temp$4)
                                         _
```

Nun ist das Untermuster `Cons(_, _)` vollständig eingetragen. Allerdings steht noch die Betrachtung des Untermusters `Nil` aus, bevor der Abgleich für das gesamte Muster vollständig ist. Auch der Switch-Knoten für diesen letzten Vergleich wird wiederum in dem `and`-Verweis des aktuellen Zielknotens eingetragen.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                            Nil(1, 0) → BODY
```

Damit ist das gesamte erste Muster übersetzt, und es wird ein Body-Knoten generiert, der den Rumpf des ersten Falles repräsentiert. Außerdem wird der Zielknoten wieder auf den Ausgangspunkt des Graphen zurückgesetzt, bevor begonnen wird, den zweiten Fall der `switch`-Anweisung in den Graphen einzufügen.

Eintragung des zweiten Falls

Auch für das zweite Muster `Pair(Nil, _)` muß zunächst ein `Pair`-Konstruktorknoteneintrag erfolgen. Hierzu werden ausgehend vom aktuellen Switch-Knoten entlang der `or`-Verweise alle Alternativen durchsucht, bis ein gleicher Knoten gefunden wird. Findet sich kein übereinstimmender Knoten, wird der neue Muster-Knoten als Alternative an das Ende der Liste gehängt. Für den `Pair`-Knoten gibt es bereits einen Eintrag, weshalb die einzige Aktion lediglich darin besteht, den Zielknoten auf diesen `Pair`-Knoten zu setzen.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                            Nil(1, 0) → BODY
```

Nun verbleiben die beiden Teilmuster `Nil` und `_`. Der Konstruktor-Knoten für `Nil` wird wiederum nach dem gleichen Schema in der Switch-Leiste eingetragen, auf die der `and`-Verweis des aktuellen Zielknotens zeigt. Diesesmal existiert lediglich ein Konstruktor-Knoten für `Cons`, weswegen `Nil` an das Ende der Liste gehängt wird.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                            Nil(1, 0) → BODY
                 Nil(1, 0)
```

Bevor der Body-Knoten für den zweiten Fall eingetragen werden kann, muß noch der DefaultPat-Knoten für das letzte zu betrachtende leere Untermuster eingefügt werden.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                           Nil(1, 0) → BODY
                 Nil(1, 0) → SWITCH(List, temp$1.snd, temp$6)
                                     _→ BODY
```

Eintragung des dritten Falls

`Pair(Cons(Object h1, List t1), Cons(Object h2, List t2))` wird als drittes Muster der `switch`-Anweisung analog eingetragen. Zunächst wird im aktuellen Switch-Knoten nach einem `Pair`-Konstruktor-Knoten gesucht. Die Suche ist erfolgreich und die Transformation wird an der durch den `Pair`-Knoten referenzierten inneren Switch-Leiste fortgesetzt. Hier muß nun nach einem `Cons`-Knoten gesucht werden. Dieser wird gefunden, weshalb auch hier wiederum keine Neueintragung notwendig ist. Nun stehen noch die beiden Variablen-Muster `(Object h1)` und `(List t1)` an, bevor das Untermuster `Cons(Object h1, List t1)` vollständig bearbeitet ist. Auch beim Einfügen dieser beiden Muster ändert sich am obigen Graphen nichts, da bereits zwei DefaultPat-Knoten an dieser Stelle existieren.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                           Nil(1, 0) → BODY
                 Nil(1, 0) → SWITCH(List, temp$1.snd, temp$6)
                                     _→ BODY
```

Jetzt gilt es das Untermuster `Cons(Object h2, List t2)` zu betrachten. In der aktuellen Switch-Leiste gibt es bisher lediglich einen Eintrag für `Nil`. Also wird ein `Cons`-Knoten als Alternative eingetragen.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                 Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                                _→ SWITCH(List, temp$2.tail, temp$4)
                                      _→ SWITCH(List, temp$1.snd, temp$5)
                                           Nil(1, 0) → BODY
                                           Cons(0, 2)
                 Nil(1, 0) → SWITCH(List, temp$1.snd, temp$6)
                                     _→ BODY
```

Nachdem die beiden verbleibenden Muster `(Object h2)` und `(List t2)` transformiert wurden, ist das gesamte Muster eingefügt und es wird der zu diesem Fall gehörige Body-Knoten erzeugt. Damit ist der Zwischencode für die `switch`-Anweisung aus Programm 3.4 vollständig.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
               Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                           _→ SWITCH(List, temp$2.tail, temp$4)
                                 _→ SWITCH(List, temp$1.snd, temp$5)
                                       Nil(1, 0) → BODY
                                       Cons(0, 2)→ SWITCH(Object, temp$5.head, temp$7)
                                                   _→ SWITCH(List, temp$5.tail, temp$8)
                                                         _→ BODY
               Nil(1, 0) → SWITCH(List, temp$1.snd, temp$6)
                                _→ BODY
```

Sonderfälle

Anhand des Beispiels sollte das grundsätzliche Übersetzungsprinzip klar geworden sein. Es müssen allerdings noch einige Sonderfälle diskutiert werden, die in Zusammenhang mit DefaultPat-Knoten entstehen können. Hierzu wird folgende `switch`-Anweisung betrachtet:

```
switch (e) {
   case Pair(Nil, Nil): ...
   case Pair(_, Cons(_, _)): ...
   case Pair(Cons(_, _), _): ...
}
```

Die Fallunterscheidung ist vollständig und keines der Muster wird von vorherigen Mustern vollständig überdeckt. Die Zwischendarstellung sieht nach dem Einfügen des ersten Falls folgendermaßen aus:

```
SWITCH(ListPair, e, temp$0)
  Pair(0, 2)→ SWITCH(List, temp$0.fst, temp$1)
               Nil(1, 0)→ SWITCH(List, temp$0.snd, temp$2)
                             Nil(1, 0) → BODY
```

Beim Eintragen des zweiten Falls entsteht eine Situation, in der ein DefaultPat-Knoten in eine Switch-Leiste eingetragen werden muß, in der bereits ein Konstruktor-Knoten für `Nil` steht. Ein Vorgehen nach dem beschriebenen Schema würde folgende inkorrekte Zwischendarstellung zur Folge haben:

```
SWITCH(ListPair, e, temp$0)
  Pair(0, 2)→ SWITCH(List, temp$0.fst, temp$1)
               Nil(1, 0) → SWITCH(List, temp$0.snd, temp$2)
                             Nil(1, 0) → BODY
               _         → SWITCH(List, temp$0.snd, temp$3)
                             Cons(0, 2)→ SWITCH(Object, temp$3.head, temp$4)
                                         _→ SWITCH(List, temp$3.tail, temp$5)
                                               _→ BODY
```

Hätte der Selektor `e` als Wert `Pair(Nil, Cons(`x`, `xs`))`, so würde man sich beim Interpretieren des Graphen bei dem fett gedruckten Switch-Knoten für den `Nil`-Fall entscheiden. Beim nächsten Switch-Knoten müßte man dann allerdings feststellen, daß jetzt nur

ein weiterer `Nil`-Knoten zur Auswahl steht, auf welchen der `Cons`-Wert nicht paßt. Der Musterabgleich würde fehlschlagen, obwohl der Wert von `e` durch den zweiten Fall abgedeckt ist. Man darf DefaultPat-Knoten also niemals in eine Switch-Leiste zusammen mit Konstruktor-Knoten einfügen, da sich sonst die Fälle nicht gegenseitig ausschließen. Man muß den DefaultPat-Knoten in eine eigene Switch-Leiste verpacken, die textuell auf der gleichen Ebene wie die bereits existierende Switch-Leiste steht. In der korrekten Lösung ist die neue Switch-Leiste über den `and`-Verweis der existierenden Switch-Leiste erreichbar. Im folgenden Graphen ist dieser Verweis nur implizit über ein bündiges Einrücken des neuen Switch-Knotens angedeutet.

```
SWITCH(ListPair, e, temp$0)
  Pair(0, 2)→ SWITCH(List, temp$0.fst, temp$1)
                Nil(1, 0) → SWITCH(List, temp$0.snd, temp$2)
                              Nil(1, 0) → BODY
              SWITCH(List, temp$0.fst, temp$1)
                _         → SWITCH(List, temp$0.snd, temp$3)
                              Cons(0, 2)→ SWITCH(Object, temp$3.head, temp$4)
                                             _→ SWITCH(List, temp$3.tail, temp$5)
                                                   _→ BODY
```

Beim Einfügen des dritten Musters `Pair(Cons(_, _), _)` darf man jetzt den `Cons`-Knoten nicht in der fett gedruckten Switch-Leiste unterbringen. Eingefügt werden darf stets nur in der untersten Switch-Leiste, da sonst die Muster nicht in der korrekten Reihenfolge von oben nach unten verglichen werden. Dagegen sind Umordnungen innerhalb einer Switch-Leiste beliebig möglich, da sich die Fälle hier gegenseitig ausschließen. Aber auch in der untersten Leiste ist der neue `Cons`-Knoten fehl am Platz, da hier der DefaultPat-Knoten sämtliche weitere Konstruktor-Knoten verbietet. Es muß deswegen konsequenterweise ein weiterer Switch-Knoten für den `Cons`-Musterabgleich angelegt werden. Die vollständige Zwischenrepräsentation sieht damit folgendermaßen aus:

```
SWITCH(ListPair, e, temp$0)
  Pair(0, 2)→ SWITCH(List, temp$0.fst, temp$1)
                Nil(1, 0) → SWITCH(List, temp$0.snd, temp$2)
                              Nil(1, 0) → BODY
              SWITCH(List, temp$0.fst, temp$1)
                _         → SWITCH(List, temp$0.snd, temp$3)
                              Cons(0, 2)→ SWITCH(Object, temp$3.head, temp$4)
                                             _→ SWITCH(List, temp$3.tail, temp$5)
                                                   _→ BODY
              SWITCH(List, temp$0.fst, temp$1)
                Cons(0, 2)→ SWITCH(Object, temp$1.head, temp$4)
                              _→ SWITCH(List, temp$1.tail, temp$5)
                                    _→ SWITCH(List, temp$0.snd, temp$2)
                                          _→ BODY
```

Zwischencode wie dieser entsteht immer dann, wenn sich Muster gegenseitig überlappen. Der aus solchen Repräsentationen abgeleitete Code ist meist nicht optimal. Es wäre möglich, die Fälle so zu Partitionieren, daß keine gegenseitigen Überlappungen mehr existieren. Allerdings ist dies nur möglich, wenn man hierfür Rümpfe dupliziert. Im schlimmsten Fall entstehen dabei soviele Rümpfe, wie Werte für den Selektortyp existieren.

```
PatternNode enter(Tree actual, Definition formal, PatternNode target, Switch header, Env env) {
   Tree[]       actuals = patArgs(actual);                       // Untermuster extrahieren
   Definition[] formals = patFields(actual);                     // Formale Parameterdefinitionen ermitteln
   if ((header != null) && (target.and == null)) {               // bei Bedarf ersten Switch-Knoten erzeugen
      target.and = header =
         makeSwitch(formal.type,
            makeTree(header.temp, formal.name), newTemp());
      return enter(actuals, formals, header.or =
            makeNode(actual, header, env), header, env);
   }
   header = (Switch)((header == null) ? target : target.and);  // untersten Switch-Knoten ermitteln
   while (header.and != null)
      header = (Switch)header.and;
   target = header.or;
   PatternNode node = makeNode(actual, header, env);            // Knoten aus Muster erzeugen
   if (isDefaultPat(target) ^ isDefaultPat(node))               // Knoten gehört in eigenen Switch-Knoten
      return enter(actuals, formals, (header = (Switch)(header.and =
               cloneSwitch(header))).or = node, header, env);
   while (true)                                                  // Suche in aktueller Switch-Leiste
      if (samePat(target, node))                                 // gleicher Knoten gefunden
         return enter(actuals, formals, target, header, env);
      else
      if (target.or == null)                                     // kein Knoten gefunden → Knoten einfügen
         return enter(actuals, formals,
                     target.or = node, header, env);
      else
         target = target.or;
}

PatternNode enter(Tree[] actuals, Definition[] formals, PatternNode target, Switch header, Env env) {
   for (int i = 0; i < actuals.length; i++)
      target = enter(actuals[i], formals[i], target, header, env);
   return target;
}

PatternNode makeNode(Tree pattern, Switch header, Env env) {
   if (pattern.isConstant())
      return makeConstantPat(pattern.constValue());             // Konstante
   else
      switch (pattern) {
         case Blank():
            return makeDefaultPat(header.type);                  // _, default:
         case VarDecl(Name name, _, Tree vtype, _, _):
            env.bind(name, vtype.type, header.type);
            if (sameType(vtype.type, header.type))
               return makeDefaultPat(header.type);               // (Type v)
            else
               return makeConstrPat(vtype.type);                 // (Case v)
         case Apply(Tree fn, _):
            return makeConstrPat(fn.type);                       // Case(p1, ..., pn)
         default:
            return makeConstrPat(pattern.type);                  // Case
      }
}
```

Programm 3.5: Pattern Matching-Transformationsalgorithmus

Programm 3.5 zeigt den vollständigen Algorithmus. Die Methode `makeNode` bildet dabei Muster auf ihre entsprechenden Knoten der Zwischendarstellung ab. Mit `patArgs` werden Untermuster aus einem Muster extrahiert und mit `patFields` alle Felder einer Varianten eines algebraischen Typs bestimmt. Die `make`...-Methoden sind Factory-Methoden für verschiedene Datentypen.

3.6.2 Generierung von Java-Code

Aus der Zwischenrepräsentation läßt sich auf einfache Art und Weise ein äquivalentes Java-Programm ableiten. Abhängig vom Typ eines Switch-Knotens wird eine Fallunterscheidung entweder als `switch`-Anweisung oder als verschachtelte `if`-Anweisungen implementiert. Bei algebraischen Typen vergleicht man Tags, bei Basistypen werden die Werte über den `==`-Operatoren bzw. implizit in einer `switch`-Anweisung verglichen und bei Strings, die in Konstanten-Mustern vorkommen können, wird die `equals`-Methode angewendet. Ein Body-Knoten wird durch zwei Anweisungssequenzen repräsentiert. Die erste Sequenz, der Binding Code, ist generiert und enthält alle Variablendeklarationen, die im Muster enthalten sind. Die zweite Anweisungssequenz entspricht dem Rumpf des Falls im ursprünglichen Pattern Matching-Ausdruck. Dieser muß bei `break`-Anweisungen leicht modifiziert werden, so daß sich diese Anweisungen auch auf die äußerste Fallunterscheidung im generierten Code beziehen. 3.6 zeigt den aus der endgültigen Zwischendarstellung von Beispielprogramm 3.4 erzeugten Java-Code. Man beachte, daß Switch-Leisten mit nur einem einzigen DefaultPat-Knoten bei der Übersetzung einfach wegfallen.

Dadurch daß man zu Beginn der Codeerzeugung die Struktur einer Fallunterscheidung bereits vollständig kennt, ist es relativ einfach, guten Code zu erzeugen. In der Übersetzung 3.6 wird die äußerste Switch-Leiste beispielsweise nicht durch die sonst üblicherweise für Tags verwendete `switch`-Anweisung implementiert, sondern mittels einer billigeren `if`-Anweisung. Ein intelligenterer Code-Erzeuger könnte die Information verwenden, daß die algebraischen Typen `final` deklariert sind, um die äußerste `if`-Anweisung ganz und gar wegzuoptimieren, da es keine alternativen Möglichkeiten gibt. Die inneren `switch`-Anweisungen über die beiden Tags 0 und 1 könnte man aus dem gleichem Grund durch ein `if-else`-Konstrukt ersetzen.

Die Code-Erzeugung ist soweit unproblematisch, es gibt jedoch einige Feinheiten zu beachten, um korrekten Java-Code zu erhalten. Die in Programm 3.6 kursiv gedruckte `break`-Anweisung wäre beispielsweise illegal, wenn man statt des zweiten Falls `case 0` einfach einen für dieses Beispiel äquivalenten `default`-Fall generiert hätte. In diesem Fall wäre die kursive `break`-Anweisung laut [GJS96] nicht erreichbar und damit unzulässig. Diese Problem löst man am einfachsten dadurch, daß man alle möglichen `break`-Anweisungen vorläufig generiert und in einer anschließenden Konsistenzprüfung alle generierten und nicht erreichbaren Anweisungen löscht.

Wie bereits in einem Abschnitt zuvor erwähnt, ist die in 3.6 kursiv gedruckte `throw`-Anweisung am Ende der vollständigen Fallunterscheidung notwendig, um ein korrektes Java-Programm zu erhalten.

```
List zip(List xs, List ys) {
   final ListPair temp$0 = ListPair.Pair(xs, ys);
   if (temp$0.$tag == 0) {
      final ListPair.Pair temp$1 = (ListPair.Pair)temp$0;
      switch (temp$1.fst.$tag) {
         case 0: {
            final List.Cons temp$2 = (List.Cons)temp$1.fst;
            switch (temp$1.snd.$tag) {
               case 1:
                  return Nil;
               case 0: {
                  final List.Cons temp$5 = (List.Cons)temp$1.snd;
                  Object h1 = temp$2.head;
                  List   t1 = temp$2.tail;
                  Object h2 = temp$5.head;
                  List   t2 = temp$5.tail;
                  return Cons(new Object[]{h1, h2}, zip(t1, t2));
               }
            }
            break;
         }
         case 1:
            return List.Nil;
      }
      throw new Error();
   }
}
```

Programm 3.6: Generierter Java-Code für `zip` aus Programm 3.4

3.6.3 Vollständigkeitsprüfung für Fallunterscheidungen

Die Überprüfung, ob eine Fallunterscheidung vollständig ist, d.h. alle möglichen Werte durch Muster abgedeckt werden, ist vor allem dann keine einfache Angelegenheit, wenn die Muster sich gegenseitig überdecken. Es scheint vor allem auch kein intuitives Vorgehen zu geben, über welches man die Eigenschaft überprüfen kann, ohne konkret für alle Werte zu überprüfen, ob es ein überdeckendes Muster gibt. Bisherige Veröffentlichungen über die Übersetzung von Pattern Matching [Aug85, PJ86, FH88] geben keine Lösung für das Problem an.

3.6.3.1 Darstellung von Wertemengen

Dieser Abschnitt stellt ein Verfahren vor, bei welchem aus der Zwischendarstellung für Pattern Matching-Ausdrücke ein Baum generiert wird, welcher alle durch die Muster abgedeckten Werte repräsentiert.[11] Der Wertebaum ist von der Struktur her fast identisch zur Zwischenrepräsentation. Eine Fallunterscheidung in der Zwischenrepräsentation entspricht einem Teilbaum, der die Menge von Werten charakterisiert, die durch die Fälle der Fallunterscheidung abgedeckt werden. Konkret wird ein Wertebaum stets aus den folgenden drei verschiedenen (inneren) Knotentypen aufgebaut:

[11] Man beachte, daß die Wertemengen im allgemeinen unendlich groß sind, aber durch einen endlichen Wertebaum repräsentiert werden müssen.

( ) *Vollständige Knoten* repräsentieren vollständige Wertemengen. Jeder Knoten geht aus einer Fallunterscheidung der Zwischenrepräsentation hervor. Vollständige Fallunterscheidungen entsprechen vollständigen Knoten im Wertebaum. Ein vollständiger Knoten repräsentiert also alle Werte des Typs der Fallunterscheidung, die mit dem Knoten assoziiert ist. Vollständige Knoten besitzen höchstens einen Nachfolger.

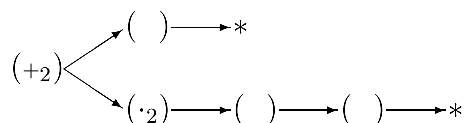
$(+_n)$ *Summen-Knoten* repräsentieren Werte, die mit Hilfe von n verschiedenen Konstruktoren aufgebaut werden können. Ein Summen-Knoten $(+_n)$ hat genau n geordnete Nachfolger ($n \geq 2$). Jeder Nachfolger ist ein Produkt-Knoten.

$(\cdot_n)$ *Produkt-Knoten* beschreiben Werte, die durch einen bestimmten n-stelligen Konstruktor ($n \geq 1$) konstruiert werden können. Konstruktoren mit $n = 0$ werden durch vollständige Knoten repräsentiert. Ein Produkt-Knoten hat genau einen Nachfolger. Dieser repräsentiert die für das erste Feld des Konstruktors möglichen Werte. Die restlichen Felder werden der Reihe nach konkateniert.

Die Blätter des Baums werden mit $*$ markiert. Sie haben sonst keine Bedeutung. Betrachten wir zunächst eine einfache Fallunterscheidung für den algebraischen `List`-Typ.

```
switch (e) {                  SWITCH(List, e, temp$0)
   case Nil: ...                Nil(1, 0) → BODY
   case Cons(_, _): ...         Cons(0, 2)→ SWITCH(Object, e.head, temp$1)
}                                            _→ SWITCH(List, e.tail, temp$2)
                                                  _→ BODY
```

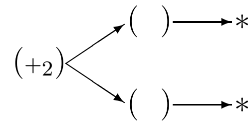
Die Werte, die durch die Muster abgedeckt werden, lassen sich mit Hilfe der Zwischenrepräsentation leicht in die Form eines Wertebaums übertragen. Der erste Switch-Knoten bestimmt, daß Werte über den `Nil`- und den `Cons`-Konstruktor konstruiert werden. Die beiden nächsten Switch-Knoten lassen beliebige Werte als `Cons`-Felder zu. Ein dazu passender Wertebaum sieht also folgendermaßen aus:



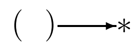
3.6.3.2 Äquivalente Wertebäume

Wie kann man an dem obigen Wertebaum erkennen, ob alle möglichen `List`-Werte abgedeckt sind, die ursprüngliche Fallunterscheidung also vollständig ist? Betrachten wir zunächst einmal den Unterbaum $(\cdot_2) \longrightarrow (\ ) \longrightarrow (\ )$. Dieser beschreibt Werte, die mittels des `Cons`-Konstruktors erzeugt werden. Die vollständigen Knoten des Unterbaums geben an, daß beliebige Parameter-Werte des Konstruktors abgedeckt sind. Der Unterbaum beschreibt also die Menge aller Werte, die mit dem `Cons`-Konstruktor konstruiert

werden können. Anders ausgedrückt: Diese Wertemenge ist vollständig. Damit ist der Unterbaum äquivalent zu einem einzigen vollständigen Knoten. Man erhält folgenden äquivalenten Wertebaum:



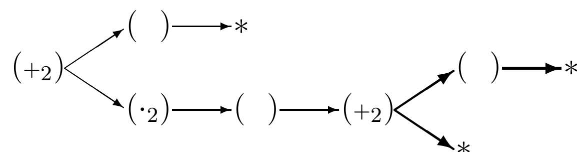
Der Baum besagt, daß über beide `List`-Konstruktoren `Nil` und `Cons` vollständige Wertemengen aufgebaut werden. Da es nur diese beiden Konstruktoren für den Datentyp `List` gibt, können damit also auch beliebige `List`-Werte konstruiert werden. Der angegebene Baum ist also äquivalent zu einem einzigen vollständigen Knoten.



Dieser minimale Baum repräsentiert alle möglichen Listen, die konstruiert werden können. Damit wurde bewiesen, daß die ursprüngliche `switch`-Anweisung eine vollständige Fallunterscheidung durchführt. Betrachten wir nun kurz eine nicht-vollständige Fallunterscheidung:

```
switch (e) {                  SWITCH(List, e, temp$0)
   case Nil: ...                Nil(1, 0) → BODY
   case Cons(_, Nil): ...       Cons(0, 2)→ SWITCH(Object, e.head, temp$1)
}                                            _→ SWITCH(List, e.tail, temp$2)
                                                  Nil(1, 0) → BODY
```

Der dazugehörige Wertebaum sieht so aus:



Wie man leicht sieht, ist der fett gedruckte Unterbaum nun nicht mehr äquivalent zu einem vollständigen Knoten, da überhaupt keine `Cons`-Werte enthalten sind. Der gesamte Wertebaum ist damit auch nicht äquivalent zu einem einzigen vollständigen Knoten. Somit kann die ursprüngliche Fallunterscheidung auch nicht vollständig sein.

3.6.3.3 Erzeugung eines Wertebaums

Mit den einführenden Beispielen sollte bereits die Vorgehensweise beim Test auf vollständige Fallunterscheidungen klargeworden sein. Die Vollständigkeitsprüfung für Fallunterscheidungen läuft demnach in zwei Schritten ab:

1. Erzeugung eines Wertebaums aus der Zwischenrepräsentation
2. Reduktion des Wertebaums

In der ersten Phase wird über die Zwischenrepräsentation ein Wertebaum konstruiert, auf welchen in der zweiten Phase ein Reduktionsalgorithmus angewendet wird. Dieser vereinfacht den Baum sukzessive dadurch, daß Unterbäume, die eine vollständige Wertemenge beschreiben, durch einen vollständigen Knoten ersetzt werden. Läßt sich der Wertebaum auf einen einzigen vollständigen Knoten reduzieren, so ist die Fallunterscheidung vollständig. Programm 3.7 zeigt den rekursiven Algorithmus zur Erzeugung eines Wertebaums aus der Zwischenrepräsentation eines Pattern Matching-Ausdrucks.

```
void toValueTree(PatternNode pat, Node target) {
   switch (pat) {
      case Switch(_, _, _):
         if (isFinalAlgebraicType(pat.type)) {
            Switch header = (Switch)pat;
            if (onlyDefaultPat(pat))
               target = target.appendCompl();
            else
               target = target.appendSum(numConstr(pat.type));
            do {
               while ((pat = pat.or) != null)
                  toValueTree(pat, target);
               pat = header = header.and;
            } while (header != null);
         }
         else if ((pat = defaultCase(pat)) != null)
            toValueTree(pat.and, target.appendCompl());
         else
            target.appendSum(1);
         break;
      case ConstrPat(int tag, int args):
         toValueTree(pat.and, target.appendProd(tag, args));
         break;
      case DefaultPat():
         if (target.complete())
            toValueTree(pat.and, target);
         else {
            PatternNode[] pats = expand(pat);
            for (int i = 0; i < constrs.length; i++)
               toValueTree(pats[i], target.appendProd(i, constrArgs(pats[i])));
         }
   }
}
```

Programm 3.7: Ermittlung eines Wertebaums aus der Zwischenrepräsentation

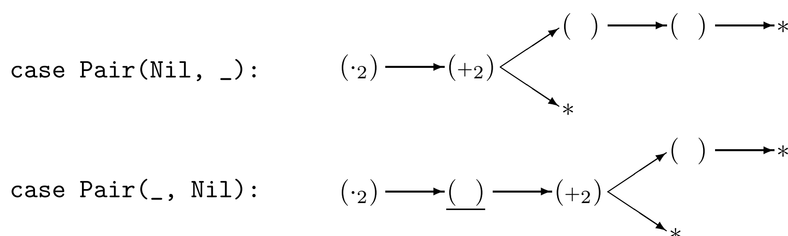
In Programm 3.7 werden mit dem Typ `Node` die Knoten des Wertebaums dargestellt. Im Prinzip traversiert der Algorithmus die Zwischendarstellung und bildet jeden Switch-Knoten, der eine Fallunterscheidung über einen Typ mit n Varianten repräsentiert, durch einen Summen-Knoten $(+_n)$ ab. Die einzelnen Fälle der Fallunterscheidung werden dann als entsprechende Nachfolger dieses Summenknotens eingetragen. Konkret wird ein Konstruktor-Knoten für Tag i als i-ter Nachfolger in Form eines Produkt-Knotens dargestellt. Besitzt eine Fallunterscheidung in der Zwischenrepräsentation lediglich einen Default-Eintrag, kann der Switch-Knoten gleich auf einen vollständigen Knoten im Wertebaum abgebildet werden. Die Abbildung von DefaultPat-Knoten ist etwas aufwendiger. Muß ein DefaultPat-Knoten in einer durch einen Summen-Knoten repräsentierten Fallunterscheidung eingetragen werden, hat folgende Expansion zu erfolgen:

$$_ \Longrightarrow \begin{array}{c} Konstruktor_1(_, \ldots, _) \\ \ldots \\ Konstruktor_n(_, \ldots, _) \end{array}$$

wobei die $Konstruktor_i$ die Konstruktoren des zum leeren Muster gehörigen algebraischen Typs sind. Dies bedeutet, daß anstelle eines leeren Musters alle möglichen Konstruktor-Muster mit „leeren Feldern“ eingetragen werden. Dies entspricht dem intuitiven Verständnis eines leeren Musters.[12] Wichtig ist, daß diese Expansion nicht dann durchgeführt wird, wenn bereits ein vollständiger Knoten eine (vollständige) Fallunterscheidung repräsentiert. Das nächste Beispiel soll die Expansion leerer Muster nochmals verdeutlichen. Vorausgesetzt wird folgende Fallunterscheidung:

```
switch (p) {
   case Pair(Nil, _):
      ...
   case Pair(_, Nil):
      ...
}
```

Für die beiden Fälle lassen sich jeweils getrennt Wertebäume angeben. Der gesuchte Wertebaum für die gesamte Fallunterscheidung ergibt sich durch eine Verschmelzung der beiden Wertebäume, so daß damit die Vereinigungsmenge der beiden ursprünglichen Wertemengen beschrieben wird.

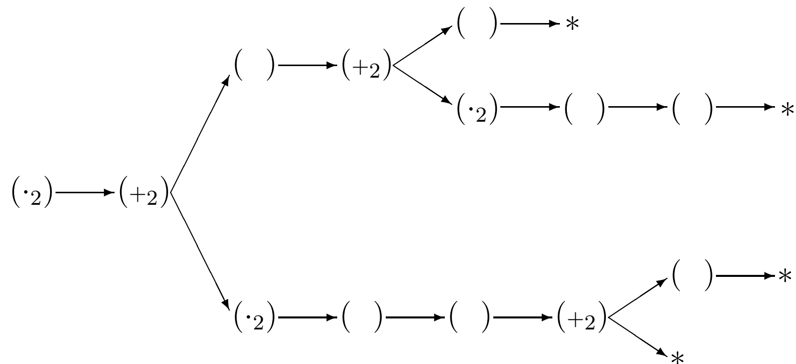


[12]Diese Expansion des leeren Musters ist gleichbedeutend mit einer Duplikation von Code bzw. Teilmustern.

Beim Verschmelzen der beiden Wertebäume muß der markierte vollständige Knoten des zweiten Falls expandiert werden, da an der gleichen Stelle im ersten Baum ein Summenknoten vorliegt. Die oben angegebene Expansionsregel sieht auf der Seite der Wertebäume in diesem Fall folgendermaßen aus: [13]

$$(\ )\longrightarrow\alpha \quad\Longrightarrow\quad (+_2)\begin{cases}(\ )\longrightarrow\alpha\\ (\cdot_2)\longrightarrow(\ )\longrightarrow(\ )\longrightarrow\alpha\end{cases}$$

Diese Regel wird bei der Verschmelzung der obigen beiden Bäume insgesamt zweimal angewendet. Es entsteht schließlich folgender Wertebaum:

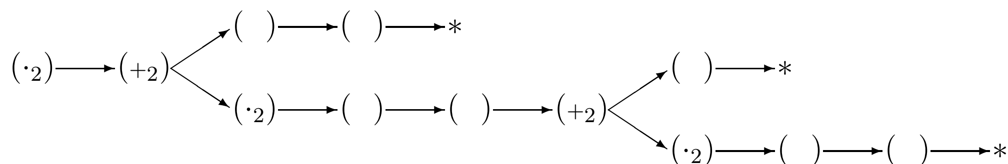


Es sei nochmals darauf hingewiesen, daß dieses Beispiel lediglich eine Veranschaulichung für das Expandieren von leeren Mustern auf der Ebene der Wertebäume darstellt. Algorithmus 3.7 arbeitet direkt auf der Zwischendarstellung und muß explizit keine Bäume verschmelzen.

Der Vollständigkeit wegen wird abschließend noch der Wertebaum für das Beispiel aus Abschnitt 3.6.1.3 angegeben. Zum Vergleich ist zuvor die zugehörige Zwischenrepräsentation des Pattern Matching-Ausdrucks abgedruckt.

```
SWITCH(ListPair, temp$0, temp$1)
  Pair(0, 2)→ SWITCH(List, temp$1.fst, temp$2)
                Nil(1, 0) → SWITCH(List, temp$1.snd, temp$6)
                                  _→ BODY
                Cons(0, 2)→ SWITCH(Object, temp$2.head, temp$3)
                              _→ SWITCH(List, temp$2.tail, temp$4)
                                    _→ SWITCH(List, temp$1.snd, temp$5)
                                          Nil(1, 0) → BODY
                                          Cons(0, 2)→ SWITCH(Object, temp$5.head, temp$7)
                                                        _→ SWITCH(List, temp$5.tail, temp$8)
                                                              _→ BODY
```

[13]Wie man in Abschnitt 3.6.3.4 sehen wird, ist dies im Prinzip die Umkehrung der Summenregel des dort angegebenen Reduktionsalgorithmus.



3.6.3.4 Reduktionsalgorithmus

Bei dem vorangehenden größeren Beispiel ist nicht mehr ohne weiteres zu sehen, ob der Baum äquivalent zu einem vollständigen Knoten ist. Die Überprüfung ob ein Wertebaum äquivalent zu einem vollständigen Knoten ist, kann allgemein mit Hilfe von zwei einfachen Reduktionsregeln beschrieben werden:

Produktregel: $$(\cdot_n)\longrightarrow\underbrace{(\ \ )\longrightarrow\ldots\longrightarrow(\ \ )}_{n\,(\ \ )\text{-Knoten}} \quad\Longrightarrow\quad (\ \ )$$

Summenregel: $$(+_n)\left\langle\begin{array}{l}(\ \ )\longrightarrow\alpha\\ \quad\vdots\\ (\ \ )\longrightarrow\alpha\end{array}\right\}\ n\,(\ \ )\text{-Knoten} \quad\Longrightarrow\quad (\ \ )\longrightarrow\alpha$$

Diese beiden Regeln geben ein indeterministisches Verfahren zur Reduktion eines beliebigen Wertebaums an. Implementieren läßt sich dieses Verfahren allerdings schlecht, da es zum einen indeterministisch ist und zum anderen, vor allem wegen der Summenregel, nur sehr ineffizient realisiert werden kann. Für alle Nachfolger eines Summen-Knotens muß hier sichergestellt werden, daß die indirekten Nachfolger α alle gleich sind. Man kann jedoch leicht einen deterministischen Algorithmus angeben, bei dem diese Überprüfung nicht mehr durchgeführt werden muß. Die Idee besteht darin, die Knoten des Wertebaums in Tiefensuchreihenfolge zu durchlaufen. Trifft man auf einen Summen- oder einen Produkt-Knoten muß die dazugehörige Reduktionsregel anwendbar sein. Ist dies nicht der Fall, kann der Wertebaum auch nicht auf einen einzigen vollständigen Knoten reduziert werden. Es ist leicht einzusehen, daß nun in der Summenregel die mit α bezeichneten Unterbäume gar nicht mehr betrachtet werden müssen, da es sich hier stets um eine gleichlange Sequenz von vollständigen Knoten handeln muß. Dies setzt natürlich voraus, daß die ursprüngliche Fallunterscheidung korrekt typisiert war.

Kapitel 4

Erweiterbare Übersetzer

Dieses Kapitel diskutiert erweiterbare Übersetzer auf der Ebene der Software-Architektur. Einführend wird zunächst ein grober Überblick über das Gebiet der Software-Architektur im allgemeinen geben. Dabei werden zentrale Begriffe definiert und in Beziehung zueinander gesetzt. Anschließend wird ein Software-Architekturmuster vorgestellt, mit welchem frei erweiterbare, hierarchisch aufgebaute Komponentensysteme konstruiert werden können. Es wird eine Variante dieses Architekturmusters beschrieben, mit der sich auf sehr flexible Art und Weise Software-Architekturen für erweiterbare Übersetzer entwerfen lassen, die dem gängigen Architekturstil eines batch-sequentiellen Repositories folgen. Abschließend wird eine konkrete Instanz dieses Architekturmusters in Form eines erweiterbaren Java-Übersetzers erläutert.

4.1 Grundkonzepte der Software-Architektur

Übersetzer sind im allgemeinen komplexe Software-Systeme. Deswegen spielt vor allem hier der Entwurf der Systemstruktur eine bedeutende Rolle. Auf dieser Ebene ist das Feld der Software-Architektur angesiedelt. Diese relativ neue Disziplin stellt die Struktur eines Systems als eine Menge von *Komponenten* dar, zwischen denen bestimmte Beziehungen bestehen [GS94, SG96]. Die Interaktionsmuster werden durch sogenannte *Konnektoren* beschrieben. Eine Komponente ist ein gekapselter Teil eines Software-Systems mit einer bestimmten Schnittstelle. Komponenten können wiederum Subsysteme darstellen, die gemäß einer bestimmten Architektur aus Unterkomponenten und Konnektoren zusammengesetzt sind. Eine Software-Architektur wird gewöhnlich unter verschiedenen Sichtweisen dargestellt, um jeweils bestimmte Eigenschaften eines Software-Systems in den Vordergrund zu rücken. So wird erkennbar, ob die Architektur eines Systems den gestellten Anforderungen gerecht wird. In [SNH95] wird vorgeschlagen, eine Software-Architektur aus vier verschiedenen Sichten zu beschreiben: aus Sicht der konzeptionellen Architektur (Komponenten, Konnektoren), der Modul-Architektur (Untersysteme,

Module), der Code-Architektur (Quelltexte, Bibliotheken) und der Laufzeit-Architektur (Threads, Prozesse).

Ein zentraler Aspekt beim Entwurf einer konkreten Software-Architektur ist die Wiederverwendung von Strukturen ähnlich aufgebauter Systeme. 1992 wurde von Perry und Wolf der Begriff des *Architekturstils* eingeführt [PW92]. Ein Architekturstil beschreibt eine Familie von Software-Systemen abhängig von vergleichbaren System-Strukturen. Beispielsweise lassen sich Pipeline-, Client-Server- oder geschichtete Architekturen als Architekturstile verstehen [SG96]. Eine Beschreibung eines Architekturstils besteht im allgemeinen aus mehreren Aspekten. Ein *Vokabular* beschreibt die Entwurfselemente, d.h. die Komponenten und Konnektoren aus denen Systeme aufgebaut werden, *Konfigurationsregeln* geben Beschränkungen für die Komposition der Elemente an und *semantische Interpretationen* legen die Bedeutung der einzelnen Entwurfselemente fest. Außerdem werden Aussagen darüber gemacht, wann ein Architekturstil Anwendung finden sollte, welche Invarianten und Spezialisierungen es gibt und welche Konsequenzen sich für Software-Systeme ergeben. Die Verwendung von Architekturstilen bringt im allgemeinen eine Vielzahl von Vorteilen mit sich [Gar95]:

- Die Wiederverwendung von Software-Entwürfen wird gefördert. Erfolgreich eingesetzte Standardlösungen erlauben es, neue Probleme auf sichere Art und Weise zu lösen.
- Die Wiederverwendung von gemeinsamem Code wird möglich.
- Ein System wird für andere Personen einfacher verständlich, wenn konventionelle Strukturen Verwendung finden. Auch Spezialisierungen eines bestimmten Architekturstils sind einfach beschreibbar, da auf ein genau definiertes Vokabular zurückgegriffen werden kann.
- Visualisierungen von Architekturen sind möglich. Mit textuellen und grafischen Darstellungen können anwendungsspezifische Details eines Entwurfs intuitiv veranschaulicht werden.

Architekturstile sind auch in Zusammenhang mit Entwurfsmustern von Bedeutung. Sie stellen eine Sprache zur Formulierung von Entwurfsmustern für Software-Architekturen zur Verfügung [M+97]. In [B+96] werden diese *Architekturmuster* als grundlegende strukturelle Organisationsschemata für Software-Systeme erklärt. Sie definieren eine Menge von Untersystemen, spezifizieren ihre Verantwortlichkeiten und geben Regeln und Richtlinien an, wie die Beziehungen zwischen den Komponenten zu gestalten sind. Damit legen sie systemweite strukturelle Eigenschaften einer Anwendung fest, die sich auch auf die Architektur ihrer Untersysteme auswirkt. Obwohl es möglich sein sollte, jeden Architekturstil durch ein oder mehrere geeignete Architekturmuster auszudrücken, dürfen die Begriffe keinesfalls gleichgesetzt werden. Wie bereits ansatzweise erwähnt, unterscheiden sich Architekturstil und Entwurfsmuster in mehrerlei Hinsicht [M+97, B+96]:

- Architekturstile beschreiben die Struktur von Anwendungen nur auf oberster Ebene, Entwurfsmuster existieren dagegen für mehrere Ebenen. Neben Architekturmustern, die die grundlegende Struktur einer Anwendung festlegen, wird die Architektur von Untersystemen oftmals mit den klassischen Entwurfsmustern aus [G+95] beschrieben.

- Architekturstile sind unabhängig voneinander. Ein Muster hängt dagegen zum einen von Untermustern ab, die es enthält bzw. mit denen es interagiert, zum anderen hat das Obermuster, in dem es eingebettet ist, Einfluß.

- Entwurfsmuster sind problemorientierter als Architekturstile. Sie zielen auf eine immer wiederkehrende Entwurfsproblematik ab und bieten eine Lösung in dem Kontext an, in dem das Problem auftritt.

- Entwurfsmuster und Architekturstile repräsentieren komplementäre Aspekte eines Entwurfs. Ein Architekturstil bietet eine Sprache zum Konstruieren von geeigneten Architekturmustern. Deren Instanzen verkörpern konkrete Software-Architekturen des zugehörigen Architekturstils. Für einen Architekturstil können verschiedene korrespondierende Entwurfsmuster mit jeweils unterschiedlichen nicht-funktionellen[1] Eigenschaften angegeben werden.

4.2 Architekturmuster *Context-Component*

4.2.1 Motivation

Die Software-Architektur beschreibt die Grobstruktur eines Systems als Kombination von verschiedenen Komponenten. Die Architektur der Komponenten läßt sich wiederum in dieser Form beschreiben. Auf der Implementationsebene ist die Zusammensetzung aus den einzelnen Komponenten vor allem bei objektorientierten Sprachen oft nur implizit bei einer Codeinspektion erkennbar. Es gibt viele verschiedene Interaktionsmuster zwischen Komponenten, so daß es recht schwierig festzustellen ist, welche Komponenten voneinander abhängen. Dies erschwert es ungemein, ein System zu verstehen oder gar zu verändern bzw. zu erweitern. Dieses Entwurfsmuster hilft bei der Implementierung eines Systems, das sich hierarchisch aus verschiedenen Subsystemen zusammensetzt. Es separiert die Komposition eines Systems aus einzelnen Komponenten von der konkreten Implementierung der Komponenten selbst. Die Komposition ist auf diese Weise explizit erkennbar. Das Muster bietet einen uniformen Weg, Systeme zu erweitern, zu re-konfigurieren und wiederzuverwenden.

[1]In [B+96] werden nicht-funktionelle Aspekte als Eigenschaften eines Systems definiert, die nicht durch die funktionelle Beschreibung des Systems abgedeckt werden. Typischerweise handelt es sich dabei um Aspekte, die mit Zuverlässigkeit, Kompatibilität, Wiederverwendbarkeit, Effizienz oder Wartbarkeit eines Systems in Verbindung stehen.

4.2.2 Idee

Die Komponenten eines Systems werden in einem *Context*-Objekt aggregiert. Jede Komponente ist genau in einen solchen Kontext eingebettet und kann nur über das Kontext-Objekt Referenzen auf die übrigen Komponenten erhalten. Hierzu bietet das Kontext-Objekt für alle Komponenten *Factory-Methoden* [G+95] an. In diesen Methoden wird ein Protokoll spezifiziert, das bestimmt, auf welche Art und Weise eine Komponente instanziiert wird. Typischerweise wird bei jedem Factory-Methoden-Aufruf entweder stets eine neue Instanz einer Komponente angelegt, oder die Komponente ist relativ zum Kontext ein *Singleton* [G+95], d.h. es gibt nur eine Instanz einer Komponente im betreffenden Kontext.

Repräsentiert eine Komponente ein komplexeres Subsystem, das sich selbst aus weiteren Unterkomponenten zusammensetzt, so ist diese Komponente in einem geschachtelten Kontext einzubetten. Dieser aggregiert alle Unterkomponenten des Subsystems. Jeder Kontext definiert deswegen zusätzlich zu den Komponenten-Factories auch Factory-Methoden für geschachtelte Kontexte. Der Kontext, in dem ein aus mehreren Komponenten zusammengesetztes Subsystem deklariert ist, wird als äußerer Kontext des darin geschachtelten Subsystem-Kontexts bezeichnet. Komponenten, die in einem geschachtelten Kontext definiert sind, können auch auf die Komponenten von äußeren Kontexten zugreifen.

4.2.3 Struktur

Wie man aus Abbildung 4.1 erkennen kann, setzt sich das Architekturmuster aus vier Arten von Klassen zusammen:

Context Die abstrakte Context-Klasse stellt die gemeinsame Oberklasse aller Kontexte dar. Sie definiert lediglich eine Referenz auf den äußeren Kontext.

Component Die abstrakte Oberklasse aller Komponenten definiert eine `init`-Methode, über welche eine Komponente unmittelbar nach der Instanziierung initialisiert wird. Über diese Methode erhält die Komponente das Kontext-Objekt, in der sie eingebettet ist. Typischerweise besorgt sich eine Komponente innerhalb der `init`-Methode Referenzen zu allen übrigen Komponenten, auf die in der Komponente zugegriffen wird.

ConcreteContext In einer konkreten Kontext-Klasse wird ein bestimmter Kontext eines Systems beschrieben. Für alle darin eingebetteten Komponenten werden Factory-Methoden deklariert, welche Referenzen auf die Komponenten liefern. Diese Methoden spezifizieren zudem ein Protokoll, welches festlegt

- ob eine Komponente relativ zum Kontext ein Singleton darstellt und

- ob die Komponente in einem geschachtelten Kontext ausgeführt wird, in welchem zusätzliche Subkomponenten vereinbart sind.

Geschachtelte Kontexte lassen sich ebenfalls über Factory-Methoden erzeugen. Diese Methoden werden jedoch lediglich innerhalb der Factory-Methoden für Komponenten mit eigenen Kontexten aufgerufen. Für jede Singleton-Komponente gibt es in der konkreten Kontext-Klasse eine Instanzvariable, die die Referenz enthält.

ConcreteComponent Mit einer konkreten Komponenten-Klasse wird eine spezifische Komponente eines Systems implementiert. Eine solche Klasse definiert eine Methode `init`, welche in der zur Komponente gehörigen Factory-Methode direkt nach der Erzeugung der Komponente aufgerufen wird. Es wird das Kontext-Objekt als Parameter übergeben. Mit der `init`-Methode besorgt sich eine Komponente Referenzen auf alle weiteren Komponenten, mit denen innerhalb der Komponente interagiert wird. Kooperierende Komponenten müssen entweder aus dem gleichen, oder einem äußeren Kontext stammen. Die `init`-Methode läßt sich überladen, so daß es möglich wird, eine Komponente in verschiedenen konkreten Kontext-Klassen zu integrieren. `init`-Methoden fungieren als eine Art Adaptor an die Kontexte, in denen eine Komponente eingebettet ist. Anhand des Szenarios aus Abbildung 4.1 wird erläutert werden, daß die Trennung von Komponenteninitialisierung und Komponentenerzeugung notwendig ist, um Zyklen im Abhängigkeitsgraph der Komponenten zu durchbrechen. Es ist also nicht möglich, den Rumpf einer `init`-Methode einfach in einen Komponenten-Konstruktor zu verlagern.

Abbildung 4.1 zeigt ein Szenario, in welchem sich ein System, das auf der obersten Ebene durch Kontext `ConcreteContext1` beschrieben wird, aus zwei Komponenten zusammensetzt: `ConcreteCompA` und `ConcreteCompB`. `ConcreteCompA` ist ein Singleton. Die Komponente `ConcreteCompB` definiert zwei lokale Unterkomponenten, welche im geschachtelten Kontext `ConcreteContext2` vereinbart werden. Dieser beschreibt also die Konfiguration der Komponente `ConcreteCompB`. Konkret wird in diesem geschachtelten Kontext eine weitere Instanz von Komponente `ConcreteCompA` – auch diesmal als Singleton –, zum anderen eine Komponente `ConcreteCompC` vereinbart. Abbildung 4.1 zeigt für die Factory-Methoden aus Kontext `ConcreteContext1` die jeweiligen Implementierungen. Für den Umgang mit Singletons ist es wichtig, zuerst die Singleton-Komponente zu erzeugen und anschließend zu initialisieren. Auf diese Weise werden Zyklen im Abhängigkeitsgraph der Komponenten durchbrochen. Hätte man zwei Singletons die gegenseitig voneinander abhängen, würde man sonst in einer Endlosschleife abwechselnd neue Komponenten erzeugen.

Um die Strukturierung eines Systems mittels einer Kontext-Hierarchie besser verdeutlichen zu können, wird eine spezielle Notation eingeführt. Abbildung 4.2 zeigt eine Veranschaulichung mittels dieser Notation für das Szenario aus Abbildung 4.1. Kontexte werden durch Balken dargestellt. Singleton-Komponenten entsprechen Boxen, die unterhalb der Balken angebracht sind. Nicht-Singleton-Komponenten werden abgehoben vom Kontext, aber mit einem Pfeil verbunden, dargestellt. Wird eine Komponente in einem

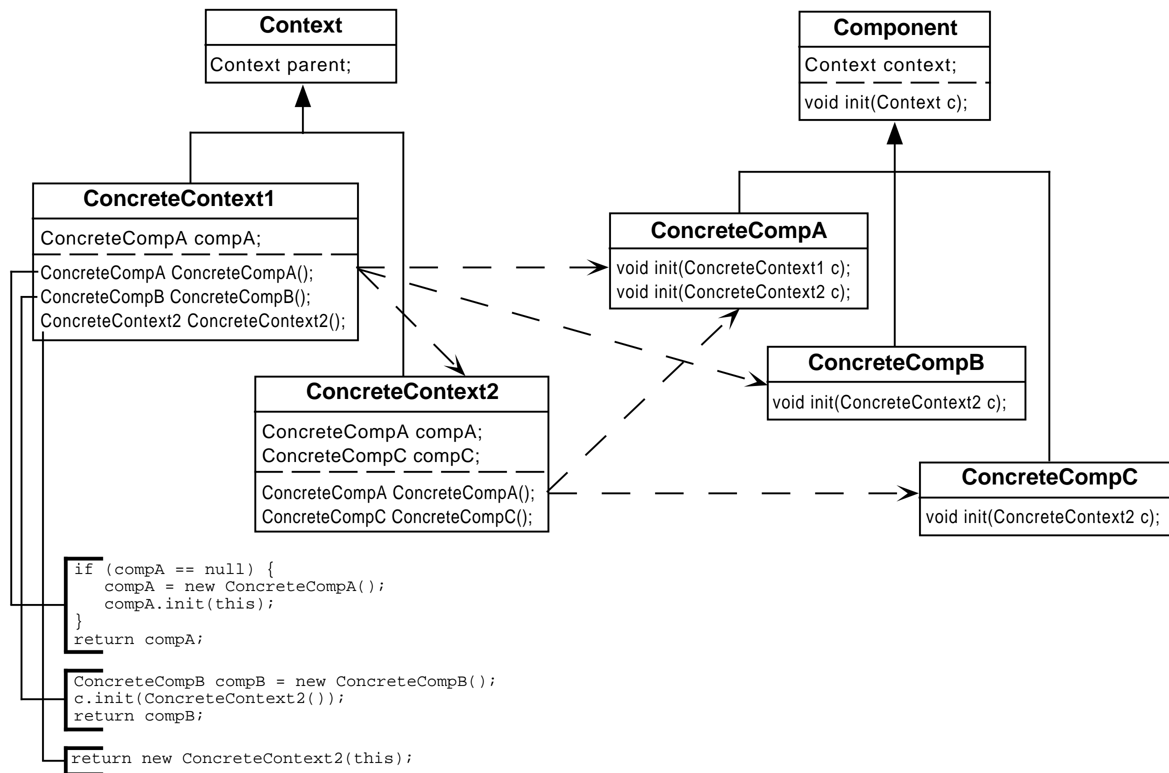


Abbildung 4.1: Struktur des Musters

geschachtelten Kontext initialisiert, so wird dieser Kontext unterhalb dieser *Trägerkomponente* gezeichnet. Geschachtelte Kontexte sind also in dieser Notationsform nur implizit erkennbar. Die initiale Instanz, die das System repräsentiert, ist in 4.1 zwar nicht modelliert, wird aber in Abbildung 4.2 der Vollständigkeit wegen als initiale Trägerkomponente eingezeichnet. In ihr wird Kontext `ConcreteContext1` instantiiert.

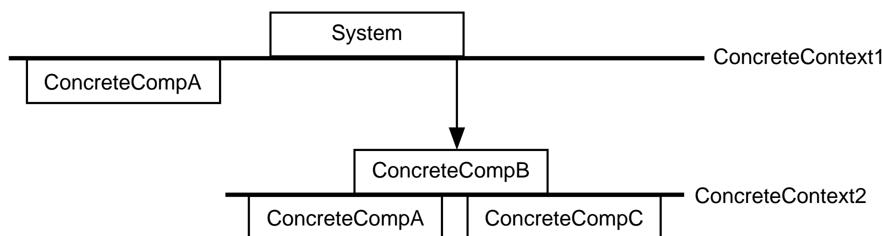


Abbildung 4.2: Schematische Notation einer Systemarchitektur

4.2.4 Konsequenzen

Kontexte unterstützen hierarchische Organisationen komplexer Systeme. Sie bieten einen uniformen Weg zur Konfiguration eines Systems. Die Zusammensetzung eines Systems

aus Komponenten erfolgt explizit und zentral in einer Klasse. Auf diese Weise dokumentieren Kontexte zugleich die strukturelle Dekomposition eines Systems. Das Context-Component-Muster kann deswegen auch als eine formale Spezifikation einer System-Architekur aufgefaßt werden. Der Aufbau eines Systems ist bereits auf der Implementationsebene gut verständlich.

Ein weiterer wichtiger Punkt ist die Entkopplung der Komposition eines Systems von der Implementierung der einzelnen Komponenten. Komponenten können aufgrund dieses Prinzips wesentlich flexibler wiederverwendet werden. Wird eine Komponente in mehreren Kontexten eingesetzt, ist jeweils lediglich ein Adaptor in Form einer Initialisierungsmethode notwendig, um die Komponente zu integrieren. Dieses Entkopplungsprinzip hat auch zur Folge, daß Konzepte, wie das des Singletons, vom Kontext und nicht von der Komponente selbst festgelegt werden.

Neben der Wiederverwendung von Komponenten in anderen Subsystemen ermöglicht das Context-Component-Muster auch die freie Austauschbarkeit von Komponenten mit der gleichen Schnittstelle, ohne daß in der Implementierung der anderen Komponenten Modifikationen vorgenommen werden müssen. Abbildung 4.3 verdeutlicht das Prinzip für die Struktur aus Abbildung 4.1.

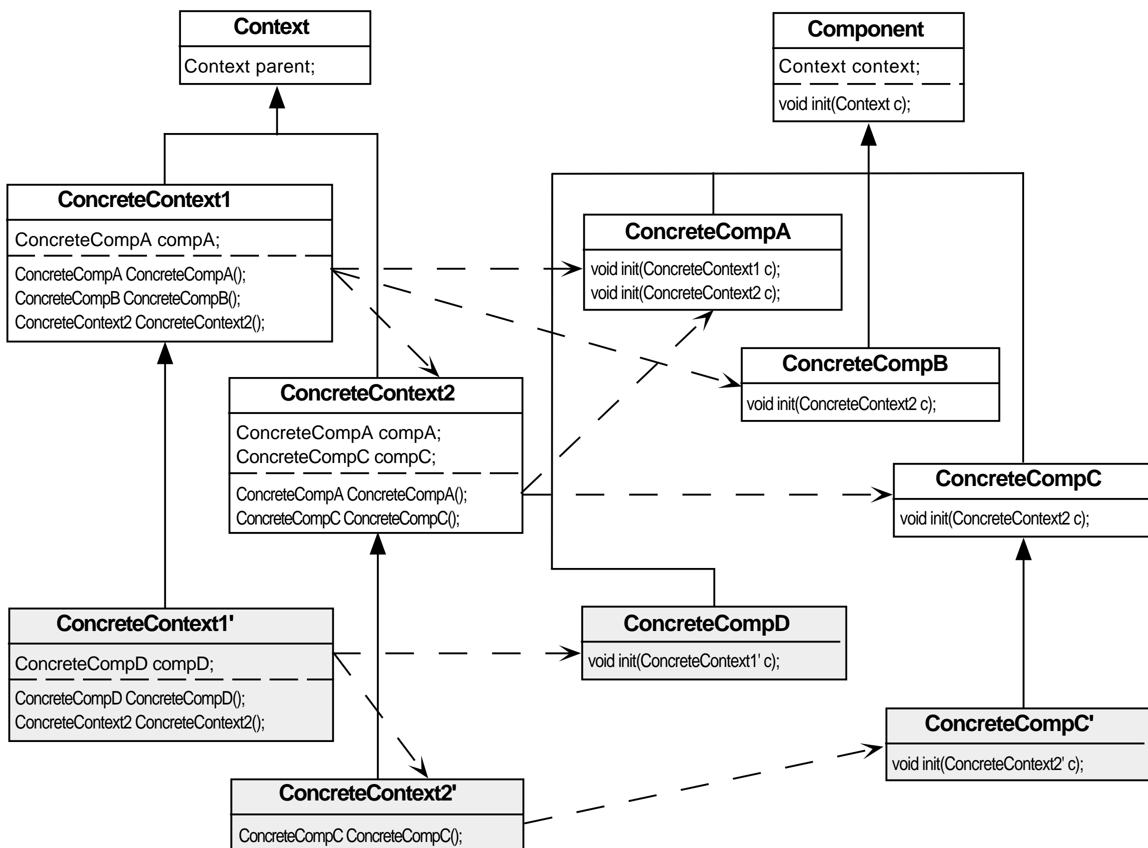


Abbildung 4.3: Erweiterung der Systemarchitektur aus Abbildung 4.1

Das durch Abbildung 4.3 beschriebene System stellt eine Erweiterung des bisherigen dar. Es wird auf oberster Ebene eine neue Komponente `ConcreteCompD` definiert. Außerdem wird im Subsystem `ConcreteCompB` die Unterkomponente `ConcreteCompC` durch `ConcreteCompC'` ersetzt. Neue Klassen werden grau unterlegt dargestellt. Es sind keine Modifikationen in bestehenden Klassen notwendig. Deswegen ist das alte System auch weiter einsetzbar, obwohl es alle seine Klassen mit dem neuen System teilt.

Ein erweitertes System erhält man einfach durch die Erweiterung einer bestehenden Kontexthierarchie, wobei Factory-Methoden so überschrieben werden, daß alte Komponenten ersetzt bzw. neue Komponenten integriert werden. Neue Systeme entstehen inkrementell aus alten, ohne diese zu zerstören. Abbildung 4.4 zeigt das Schema für das neue System in der in 4.2.3 eingeführten Notation. Die Elemente des erweiterten Systems werden in 4.4 grau dargestellt. Erweiterte Komponenten erhalten einen Schatten.

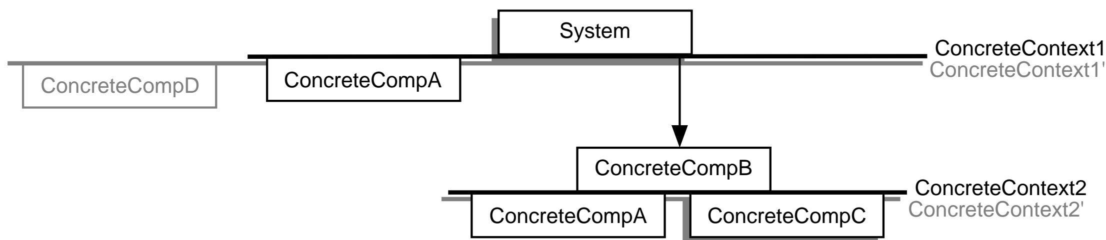


Abbildung 4.4: Architektur des erweiterten Systems

Neben den bisher erwähnten Eigenschaften bietet das Context-Component-Muster weiterhin eine Lösung für das Problem zyklischer Abhängigkeiten von Komponenten. Zyklen werden nach dem gleichen Prinzip aufgebrochen, wie das bei Java für die Klasseninitialisierung der Fall ist.

Ein entscheidender Nachteil des vorgestellten Architekturmusters besteht darin, daß beim Austauschen einer Komponente in einem geschachtelten Kontext sich die hierfür nötigen Erweiterungen der Kontexte kaskadieren. Nicht nur der innere Kontext in dem die Komponente definiert wird muß erweitert werden, sondern auch alle äußeren Kontexte, da hier jeweils die Factory-Methoden für geschachtelte Kontexte überschrieben werden müssen.

4.2.5 Implementierung

Für das Auffinden einer geeigneten strukturellen Dekomposition eines Systems in Subsysteme gelten die gleichen Richtlinien wie für das Entwurfsmuster *Whole-Part* in [B^{+}96]. Das Context-Component-Muster betont allerdings in erster Linie die Systemarchitekur. Die Implementation der Teilkomponenten ist unabhängig davon. Programm 4.1 zeigt die Kontext-Hierarchie für das Beispiel aus Abbildung 4.3 in Form eines Java-Programms.

```
abstract class Context {
   Context parent;
   Context(Context parent) {
      this.parent = parent;
   }
}

class ConcreteContext1 extends Context {
   ConcreteCompA compA;
   ConcreteContext1() {
      super(null);
   }
   ConcreteCompA ConcreteCompA() {
      if (compA == null) {
         compA = new ConcreteCompA();
         compA.init(this);
      }
      return compA;
   }
   ConcreteCompB ConcreteCompB() {
      ConcreteCompB compB = new ConcreteCompB();
      compB.init(ConcreteContext2());
      return compB;
   }
   ConcreteContext2 ConcreteContext2() {
      return new ConcreteContext2(this)
   }
}

class ConcreteContext2 extends Context {
   ConcreteCompA compA;
   ConcreteCompC compC;
   ConcreteContext2(ConcreteContext1 parent) {
      super(parent);
   }
   ConcreteCompA ConcreteCompA() {
      if (compA == null) {
         compA = new ConcreteCompA();
         compA.init(this);
      }
      return compA;
   }
   ConcreteCompC ConcreteCompC() {
      if (compC == null) {
         compC = new ConcreteCompC();
         compC.init(this);
      }
      return compC;
   }
}

class ConcreteContext1' extends Context {
   ConcreteCompD compD;
   ConcreteCompD ConcreteCompD() {
      if (compD == null) {
         compD = new ConcreteCompD();
         compD.init(this);
      }
      return compD;
   }
   ConcreteContext2 ConcreteContext2() {
      return new ConcreteContext2'(this)
   }
}

class ConcreteContext2' extends Context {
   ConcreteContext2'(ConcreteContext1' parent) {
      super(parent);
   }
   ConcreteCompC ConcreteCompC() {
      if (compC == null) {
         compC = new ConcreteCompC'();
         compC.init(this);
      }
      return compC;
   }
}
```

Programm 4.1: Kontext-Hierarchie als Java-Programm

4.2.6 Verwandte Muster

Das Context-Component-Muster ist ein allgemeines zusammengesetztes Architekturmuster, das auf eine strukturelle Dekomposition eines Systems abzielt. Ein Kontext ist eine Kombination einer *AbstractFactory* [G+95] und eines *ObjectServers*.

Whole-Part, *Composite* und *Facade* sind alternative Entwurfsmuster zur strukturellen Dekomposition eines Systems. Mit *Whole-Part* [B+96] werden Systeme mit komplexer Funktionalität aus Subsystemen mit einfacheren Funktionen aufgebaut. Ein *Whole*-Objekt aggregiert eine Anzahl kleinerer Objekte, genannt *Parts*, und baut seine Dienste auf der Funktionalität dieser Parts auf. Ein Whole-Objekt kapselt seine Parts derart, daß von außen auf sie einzeln nicht mehr zugegriffen werden kann. *Composite* [G+95] ist eine Variante von Whole-Part, bei der die Betonung auf einer uniformen Schnittstelle von zusammengesetzten Objekten und Einzel-Objekten liegt. Eine *Facade* [G+95] hilft dabei, eine einfache Schnittstelle für ein zusammengesetztes Subsystem zur Verfügung zu stellen. Eine Kapselung von Teilkomponenten findet hier nicht notwendigerweise statt. Es werden auch keine komplexen Funktionen aus einfacheren aufgebaut, sondern vorwiegend Schnittstellenanpassungen gemacht und Anfragen auf betreffende Teilkomponenten weitergeleitet.

Im Context-Component-Muster wird streng zwischen Komponenten und deren Zusammensetzung aus Subkomponenten getrennt. Keiner der genannten Entwurfsmuster macht diese Trennung. Es wird ein Mechanismus zur Verfügung gestellt, wie Subkomponenten gegenseitig aufeinander zugreifen und damit kooperieren können. Ferner ist die Erweiterbarkeit bzw. Austauschbarkeit von Komponenten Teil des Context-Component-Musters. In den oben genannten Entwurfsmustern müssen diese Mechanismen bei Bedarf auf andere Art und Weise implementiert werden.

4.3 Architekturstil *Batch-sequentielles Repository*

Wie bereits im ersten Kapitel angesprochen, lassen sich die Architekturen moderner Übersetzer mit dem Architekturstil *Repository* beschreiben [SG96]. In einem Repository müssen zwei unterschiedliche Architekturelemente unterschieden werden: Datenelemente und Operationselemente. Datenelemente definieren eine zentrale Datenstruktur, die den aktuellen Stand einer Berechnung repräsentiert. Die Operationselemente entsprechen einer Menge von unabhängigen Komponenten, die auf den globalen Daten operieren. Als Konnektoren werden gewöhnliche parametrisierte Prozeduraufrufe eingesetzt. Es lassen sich verschiedene Varianten dieses Stils identifizieren, die sich darin unterscheiden, auf welche Art und Weise bestimmt wird, in welcher Reihenfolge Komponenten auf die zentrale Datenstruktur angewendet werden.

Für Übersetzer hat sich das *Mehr-Phasen* Modell bewährt [PW92]. Die einzelnen Phasen eines Übersetzerlaufs werden in einer vordefinierten batch-sequentiellen Abfolge ausgeführt. Aus diesem Grund wird der Architekturstil eines modernen Übersetzers als

batch-sequentielles Repository bezeichnet. Strukturbäume entsprechen den Datenelementen eines Repositories und die einzelnen Phasen lassen sich als Operationselemente verstehen. Wie man im Schema von Abbildung 4.5 erkennen kann, gehören, genau genommen, zum globalen Zustand neben den Strukturbäumen auch noch eine Reihe von weiteren Datenstrukturen, auf die über globale Module zugegriffen werden kann.

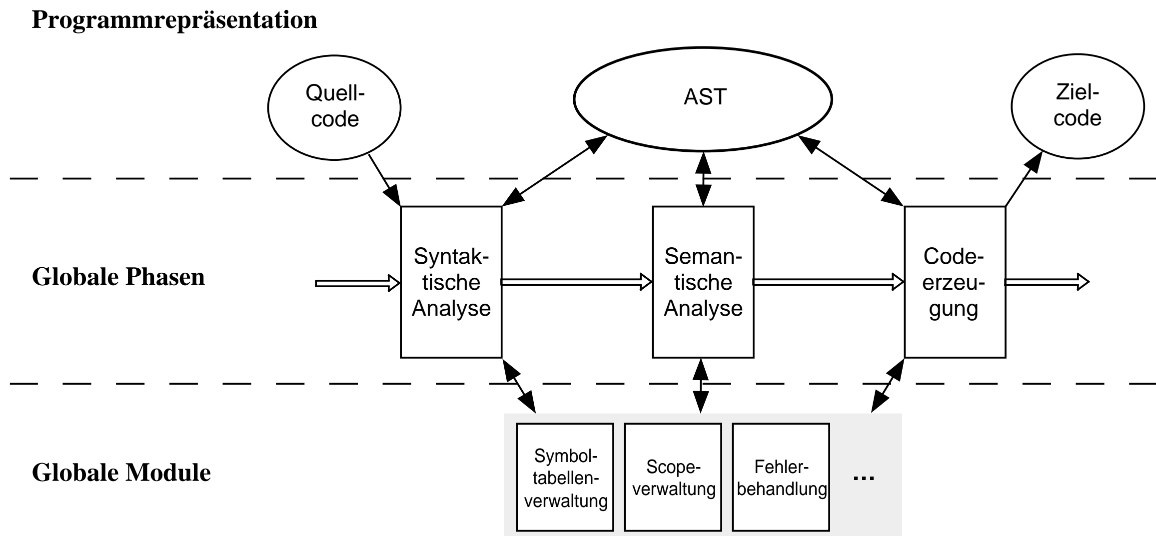


Abbildung 4.5: Übersetzer als batch-sequentielle Repositories

Eine ausführliche Beschreibung des Architekturstils eines sequentiellen Mehr-Phasen-Übersetzers kann in [PW92] nachgelesen werden.

4.4 Erweiterbare Übersetzer-Architektur

Dieses Kapitel beschreibt allgemein, wie frei erweiterbare Übersetzer aufgebaut werden können. Die in den vorangegangenen Kapiteln eingeführten Konzepte werden jetzt kombiniert. Die vorgeschlagene Übersetzer-Architektur folgt den Richtlinien des batch-sequentiellen Repository-Architekturstils. Sie basiert auf einer Variante des Context-Component-Musters. Zur Repräsentation von Datenstrukturen werden erweiterbare algebraische Typen verwendet. Erweiterbare algebraische Typen ermöglichen es, Typen und Funktionen flexibel zu erweitern, wohingegen das Context-Component-Muster einen Mechanismus beschreibt, zur Erweiterung von Modulen, die Funktionen auf solchen Datentypen anbieten. Erweiterbare algebraische Typen und das Context-Component-Muster ergänzen sich also gut, weswegen es sich auch anbietet, neben Strukturbäumen weitere übersetzerinterne Datenstrukturen mit diesen Typen zu repräsentieren. Abschnitt 4.4.5 wird auf die Repräsentation von Datenstrukturen noch näher eingehen.

4.4.1 Komponenten

Die grundlegende Entwurfsidee besteht darin, den Übersetzer aus einzelnen erweiterbaren Komponenten zusammenzusetzen, die auf gemeinsame Datenstrukturen zugreifen. Es gibt zwei verschiedene Ausprägungen von Komponenten in einem Mehr-Phasen-Übersetzer:

1. *Module*, die in Form einer Bibliothek Funktionen anbieten, welche auf Objektstrukturen eines bestimmten Datentyps operieren, die Zugriff auf eine gemeinsame (globale) Datenstruktur erlauben, oder die für Ein- und Ausgaben zuständig sind.
2. *Phasen*, die den Strukturbaum traversieren und dabei Modifikationen vornehmen bzw. Seiteneffekte erzeugen.

Module sind in dieser Definition passive Komponenten. Sie stellen lediglich eine bestimmte Funktionalität zur Verfügung. Hierzu sind oftmals Zugriffe auf andere Module notwendig. In einem Übersetzer sind Module für folgende Aufgabenbereiche vorzufinden:

1. Ein- und Ausgabe von Daten (z.B. Klassenlader, Pretty Printer, Disassembler, Fehlerausgabe)
2. Verwaltung globaler Datenstrukturen; d.h. Repräsentierung abstrakter Datentypen (z.B. Symboltabellenverwaltung, Definitionstabellenverwaltung)
3. Funktionsbibliothek für einen bestimmten Datentyp (z.B. Operationen für Typen, Konstanten, Definitionen)
4. Beschreibung eines bestimmten Quellsprachen-Aspektes (z.B. Spezifikation vordefinierter Operatoren, Modifikatoren, Typen)

Oftmals kann man ein Modul nicht unbedingt einer eindeutigen Kategorie zuordnen. Beispielsweise ist es sinnvoll, die Funktionsbibliothek für Definitionen mit der Verwaltung der Definitionstabellen zu kombinieren.

4.4.2 Dekomposition von Phasen

Im Gegensatz zu passiven Modulen, sind die Phasen die Träger der Aktivität in einem Übersetzer. Die sequentielle Anordnung der Phasen beschreibt einen Übersetzerlauf. Die Sequenz der Phasen läßt sich in mehrere Teilsequenzen mit jeweils unterschiedlicher Aufgabe unterteilen. Abbildung 4.6 verdeutlicht dies anhand eines typischen Übersetzerlaufs.

Die Gliederung in Abbildung 4.6 kann noch weiter verallgemeinert werden. Man könnte beispielsweise die *zusammengesetzten Phasen* Syntaktische Analyse und Semantische Analyse zu einer allgemeineren Phase *Frontend* aggregieren. Das Beispiel zeigt, daß sich

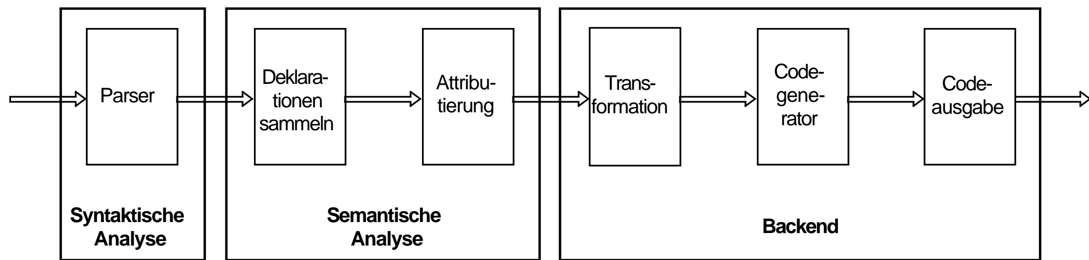


Abbildung 4.6: Gliederung eines mehr-phasigen Übersetzerlaufs

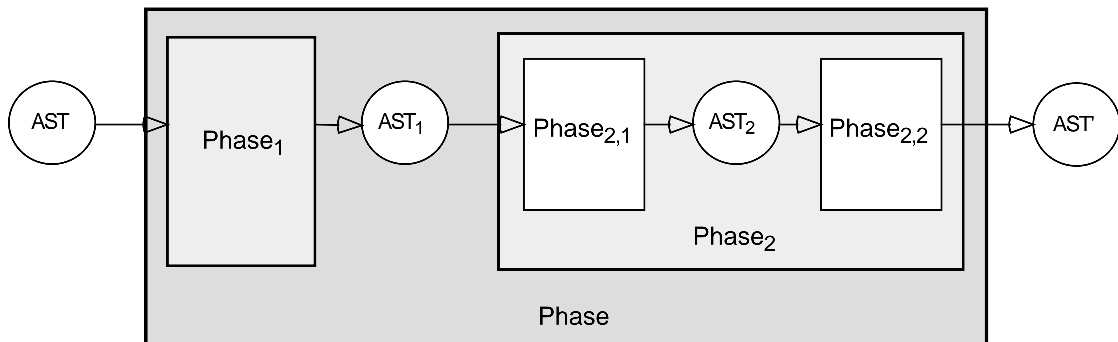


Abbildung 4.7: Geschachtelter Aufbau von Übersetzer-Phasen

die Phasen, aus denen sich ein Übersetzungsablauf zusammensetzt, gut auf hierarchische Art und Weise gliedern lassen. Abbildung 4.7 zeigt das allgemeine Prinzip, wobei hier der Strukturbaum zur Verdeutlichung mit eingezeichnet wurde.

Dieses Dekompositionsprinzip ermöglicht eine rekursive Beschreibung der Zusammensetzung eines Übersetzers aus mehreren Phasen:

1. Ein Übersetzer ist eine Phase.
2. Eine Phase ist entweder *einfach* oder *zusammengesetzt*.
3. Eine *einfache Phase* traversiert den Strukturbaum, nimmt dabei Modifikationen vor und erzeugt Seiteneffekte.[2]
4. Eine *zusammengesetzte Phase* wird selbst durch eine Sequenz von Phasen beschrieben.

Eine Zusammensetzung eines Übersetzer aus Phasen gemäß dieser rekursiven Definition bringt mehrere Vorteile mit sich. Zusammengesetzte Phasen führen neue Abstraktionsebenen ein. Diese ermöglichen es, ein System einfacher zu verstehen. Außerdem wird auf diese Weise erst eine sinnvolle Wiederverwendung von Phasen möglich. Einfache

[2]Eine einfache Phase kann auch einen Strukturbaum mehrmals traversieren. Damit ergibt sich ein weiterer Freiheitsgrad bei der Dekomposition eines Übersetzerlaufs.

Phasen dürfen oftmals nämlich nur im Kontext anderer Phasen aufgerufen werden. Sie setzen einen bestimmten Stand im Übersetzungslauf voraus. Dies drückt sich beispielsweise auch durch den Zustand globaler Datenstrukturen aus, der Voraussetzung für die Ausführung einer bestimmten Phase ist. Für den Übersetzer aus Abbildung 4.6 wäre z.B. eine Ausführung der Attributierungs-Phase nur im Kontext einer zuvor abgearbeiteten Deklarationssammel-Phase möglich; d.h. es macht nur Sinn, die zusammengesetzte Phase der semantischen Analyse als ganzes zu instantiieren bzw. wiederzuverwenden. Die Aggregierung von einzelnen Phasen zu einer allgemeineren Phase ist also Grundvoraussetzung für die Wiederverwendbarkeit von Subsystemen. Sie ermöglicht es aber auch, daß Teile des Systems im Nachhinein auf einfache Art und Weise lokal modifiziert werden können. Betrachten wir hierzu wieder das Beispiel aus Abbildung 4.6. Es wird angenommen, daß die semantische Analyse in der zusammengesetzten Transformationsphase nochmals benötigt wird; d.h. daß die Phase der semantischen Analyse darin nochmals instantiiert werden muß. Abbildung 4.8 veranschaulicht dies. Die beiden Instanzen der semantischen Analyse-Phase sind hierbei grau unterlegt dargestellt.

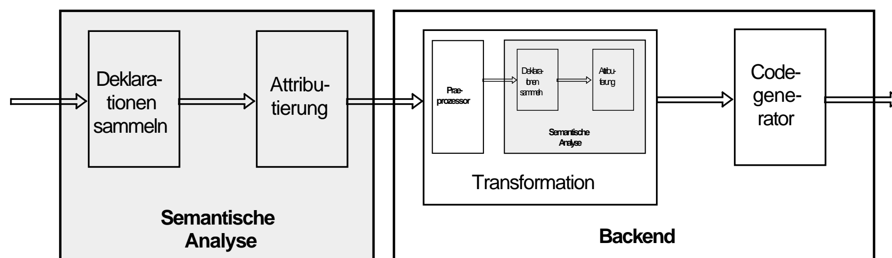


Abbildung 4.8: Mehrere Instanzen einer zusammengesetzten Phase

Erweitert man den Übersetzer nun derart, daß eine dritte Phase innerhalb der semantischen Analyse notwendig wird, so muß man diese lediglich in der Definition der zusammengesetzen semantischen Analyse-Phase einfügen. Beide Instanzen benutzen dann automatisch die modifizierte semantische Analyse-Phase. Könnte man Phasen nicht gruppieren, müßte man getrennt für beide Fälle die dritte Phase instantiieren und jeweils per Hand an der richtigen Stelle einfügen. Lokale Veränderungen haben globale Auswirkungen. Dies würde auch voraussetzen, daß alle Stellen, an denen eine Phase verwendet wird, bekannt sind.

Sinnvolle Wiederverwendungen und Modifikationen von Subsystemen eines Übersetzers sind also nur dann möglich, wenn das System in einzelne Subsysteme aufgespalten wird, die einzeln instantiierbar sind. Die folgenden beiden Abschnitte erläutern, wie die hier geschilderte Phasen-Dekomposition mit Hilfe des Context-Component-Musters implementiert werden kann.

4.4.3 Strukturelle Dekomposition von Phasen

Sowohl Module als auch Phasen müssen im Context-Component-Muster als Komponenten modelliert werden. Die rekursive Dekomposition der Phasen läßt sich leicht mit Hilfe einer Kontext-Hierarchie darstellen. Zusammengesetzte Phasen besitzen einen eigenen lokalen Kontext – sind also Trägerkomponenten in der Terminologie des Context-Component-Musters – welcher Unterphasen definiert und Module enthält, die speziell zur Ausführung dieser Phase benötigt werden. Zur Phase globale Daten lassen sich damit einfach durch lokale Module kapseln. Um zusammengesetzte Phasen besser wiederverwenden zu können, werden diese gewöhnlich nicht als Singletons modelliert. Sie können in einem Übersetzerlauf beliebig oft instantiiert werden und lassen sich damit auch in unterschiedlichen Kontexten anwenden. Einfache Phasen lassen sich dagegen meistens als Singletons in dem umgebenden Kontext einer zusammengesetzten Phase definiert. Abbildung 4.9 veranschaulicht das Entwurfsprinzip. Wie in Abschnitt 4.4.2 erwähnt, können jedoch auch kompliziertere einfache Phasen mit beispielsweise mehreren Strukturbaumtraversierungen benötigt werden. In solchen Fällen kann es durchaus sinnvoll sein, auch für einfache Phasen einen eigenen lokalen Kontext zu definieren.

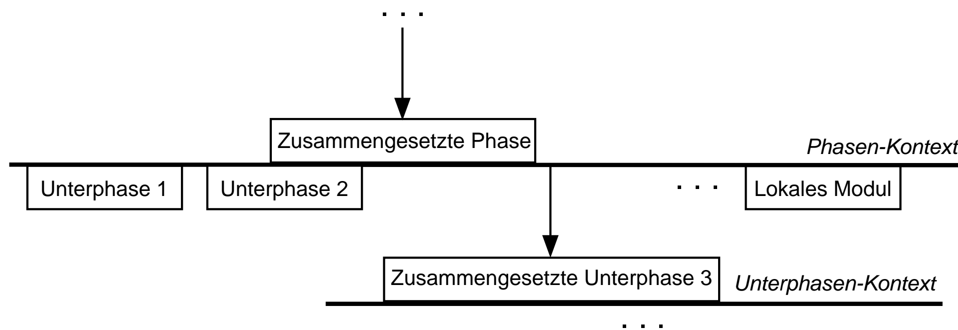


Abbildung 4.9: Strukturierung von Phasen mit dem Context-Component-Muster

4.4.4 Funktionale Dekomposition von Phasen

Mit dem Context-Component-Muster wird die statische, strukturelle Gliederung eines Übersetzers beschrieben. Die funktionale Dekomposition einzelner zusammengesetzter Phasen, d.h. die sequentielle Abfolge der Unterphasen, wird nicht erkennbar. Für die Darstellung dieses Aspekts wird eine weitere architekturelle Sichtweise benötigt.

Um zu zeigen, wie Phasen-Sequenzen gebildet werden, wird in Programm 4.2 zunächst die Klasse `TreeList` zur Repräsentation von Strukturbäumen vorgestellt. Diese Klasse besitzt eine überladene Methode `process`, mit deren Hilfe eine Phase auf die Bäume angewendet wird. Jede Phase, repräsentiert durch die Typen `TreeProcessor` für einfache und `TreeListProcessor` für zusammengesetzte Phasen, hat zudem noch eine `enter`- und eine `exit`-Methode. Diese werden zu Beginn bzw. am Ende der Phase ausgeführt.

```
class TreeList {
   ...
   TreeList process(TreeProcessor proc) throws AbortCompilation {
      TreeList trees = proc.enter(this);
      while (trees != null) {
         trees.head = proc.process(trees.head);
         trees = trees.tail;
      }
      return proc.exit(trees);
   }
   TreeList process(TreeListProcessor proc) throws AbortCompilation {
      return proc.exit(proc.process(proc.enter(this)));
   }
   ...
}
interface Processor {
   TreeList enter(TreeList trees) throws AbortCompilation;
   TreeList exit(TreeList trees) throws AbortCompilation;
}
interface TreeProcessor extends Processor {
   Tree process(Tree tree) throws AbortCompilation;
}
interface TreeListProcessor extends Processor {
   TreeList process(TreeList trees) throws AbortCompilation;
}
```

Programm 4.2: Repräsentation von Strukturbäumen und Phasen

Zur Implementierung der Phasen wird die abstrakte Oberklasse aller Komponenten `Component` aus Abschnitt 4.2.3 weiter verfeinert. Das Programm 4.3 zeigt die abstrakten Oberklassen für einfache und zusammengesetzte Phasen: `PrimitiveProcessor` und `CompositeProcessor`. Diese werden von der Klasse `DebuggableComponent` abgeleitet. `DebuggableComponent` besitzt eine überladene Methode `debug`, welche für kritische Stellen der Phase ein Debug-Verfahren implementiert. `debugId` spezifiziert die betreffende Stelle. Die Methode `debugSwitchSet` bestimmt, ob ein debug-Verfahren an dieser Stelle durchgeführt werden soll. Für erweiterbare Systeme ist es wichtig, daß es einen standardisierten und erweiterbaren Mechanismus zum Debuggen gibt. Beim Erweitern oder Wiederverwenden einer Phase ist man darauf angewiesen, daß die Debug-Methode bereits an zentralen Stellen aufgerufen wird. Nur so läßt sich durch das Überschreiben der `debug`-Methode von außen, im Nachhinein eine problemspezifische Fehlerbehandlung einfügen, ohne das Quelltextmodifikationen notwendig sind. Im Framework 4.3 wird die `debug`-Methode standardmäßig zu Beginn und am Ende einer Phase aufgerufen.

Einfache Phasen definiert man als Erweiterungen von `PrimitiveProcessor`. Dieser wendet eine Phase auf einen Strukturbaum nur dann an, wenn eine bestimmte Vorbedingung, repräsentiert durch die `needsProcessing`-Methode, zutrifft. Diese Methode entscheidet in Abhängigkeit von globalen Eigenschaften des Strukturbaums, die in einem Objekt der Klasse `CompilationEnv` gekapselt sind, ob eine einfache Phase angewendet wird. Einfache Phasen werden nach dem in Kapitel 2 für erweiterbare algebraische Typen angegebenen Muster implementiert. Die `process`-Methoden haben also eine Form wie in folgendem Beispiel:

```
abstract class Component {
   Context context;                           // der Kontext der Komponente
   void init(Context context) {               // die Komponenten-Initialisierung
      this.context = context;
   }
   abstract String getName();                 // der Name der Komponente
}
abstract class DebuggableComponent {          // Komponente die Debugging unterstützt
   boolean debug(int debugId) {
      return debugSwitchSet(getDebugName(), debugId);
   }
   boolean debug(int debugId, TreeList trees) throws AbortCompilation {
      if (debugSwitchSet(getDebugName(), debugId)) {
         if ((debugId == ENTER) || (debugId == EXIT))
            trees.process(pretty);
         return true;
      }
      return false;
   }
   ...
}
abstract class PrimitiveProcessor extends DebuggableComponent implements TreeProcessor {
   TreeList enter(TreeList treelist) throws AbortCompilation {
      debug(ENTER, treelist);
      return treelist;
   }
   TreeList exit(TreeList treelist)  throws AbortCompilation {
      debug(EXIT, treelist);
      return treelist;
   }
   Tree process(Tree tree) throws AbortCompilation {
      switch (tree) {
         case CompilationUnit(_, _, _, CompilationEnv info):
            if (needsProcessing(info))
               tree = process((CompilationUnit)tree);
            return tree;
         default:
            return tree;
      }
   }
   boolean needsProcessing(CompilationEnv info) {
      return (info.errors == 0);
   }
   abstract Tree process(CompilationUnit tree) throws AbortCompilation;
   ...
}
abstract class CompositeProcessor extends DebuggableComponent implements TreeListProcessor {
   TreeList enter(TreeList treelist) throws AbortCompilation {
      debug(ENTER, treelist);
      return treelist;
   }
   TreeList exit(TreeList treelist)  throws AbortCompilation {
      debug(EXIT, treelist);
      return treelist;
   }
   abstract TreeList process(TreeList treelist) throws AbortCompilation;
   ...
}
```

Programm 4.3: Framework für einfache und zusammengesetzte Phasen

```
Tree process(Tree tree) {
   switch (tree) {
      case ClassDecl(Name n, int mod, Tree ext, Tree[] impl, Tree[] members, _):
         ...
      case MethodDecl(Name n, int mod, _, _, _, Tree[] stats, _):
         ...
      case VarDecl(Name name, int mod, Tree type, Tree init, VarDef def):
      ...
   }
}
```

Zusammengesetzte Phasen lassen sich von der Klasse `CompositeProcessor` ableiten. An den `process`-Methoden läßt sich hier sehr einfach die Abfolge der einzelnen Unterphasen ablesen. Für den Übersetzerlauf des im Rahmen dieser Arbeit implementierten Java-Übersetzers sieht die `process`-Methode beispielsweise folgendermaßen aus:

```
TreeList process(TreeList treelist) {
   return treelist.process(context.SyntacticAnalyzer())
                  .process(context.SemanticAnalyzer())
                  .process(context.Backend())
                  .process(context.ClassWriter());
}
```

4.4.5 Repräsentation von Daten

Innerhalb eines Übersetzers gibt es eine Vielzahl von programmiersprachenspezifischen Daten, die intern durch geeignete Datenstrukturen repräsentiert werden müssen. Beispiele hierfür sind die abstrakte Syntax, Typen, Definitionen[3] oder Konstanten. Da die zu übersetzende Sprache erweiterbar sein muß, muß auch die interne Repräsentation flexibel erweitert werden können. Die Untersuchungen in Kapitel 2 haben gezeigt, daß mit erweiterbaren algebraischen Typen es zugleich möglich ist, effizient Datentypen und Operationen zu erweitern. Operationen wurden dabei von der Datentypdeklaration gesondert in eigenen Klassen bzw. Modulen vereinbart. Zur Organisation genau dieser Module eignet sich das Context-Component-Muster hervorragend. Es ermöglicht, Module flexibel zu erweitern bzw. auszutauschen und bietet einen uniformen Weg für den Zugriff auf Operationen anderer Module. Datenstrukturen, die einen Aspekt der zu übersetzenden

[3] Bei der in dieser Arbeit verwendeten Terminologie bezeichnen Definitionen, die an einen Bezeichner gebundenen Variablen, Methoden, Klassen usw. In früheren Übersetzern gab es oftmals eine einzige globale Datenstruktur, genannt Symboltabelle, in der diesbezügliche Informationen für jeden Bezeichner aufgezeichnet wurden. Deswegen werden Definitionen teilweise auch als Symbole bezeichnet. Die Symboltabelle wird heutzutage allerdings meist nur dazu verwendet, um für Bezeichner eine kompakte Codierung fester Länge zur Verfügung zu stellen. Gleiche Bezeichner werden durch die Symboltabelle auf Symbole gleicher Identität abgebildet. Die Bindungen von Bezeichnern bzw. Symbolen an Definitionen werden durch separate Definitionstabellen dargestellt. Definitionstabellen reflektieren also stets einen bestimmten Gültigkeitsbereich im zu übersetzenden Programm [WG84].

Sprache modellieren, werden deswegen innerhalb des Übersetzers jeweils durch folgende drei Elemente modelliert:

1. Erweiterbare algebraische Typen implementieren Typdefinitionen.
2. Abstract Factories liefern Instanzen des Datentyps.
3. Ein Modul bietet Funktionen an, die auf Objekten des Datentyps operieren.

Datentypen und Funktionen auf den Typen werden also separat definiert. Als Konsequenz können Datentyp und Funktionen auch getrennt voneinander erweitert bzw. modifiziert werden. Diese Flexibilität ist für erweiterbare Übersetzer äußerst bedeutsam. Es muß möglich sein, daß in unterschiedlichen Kontexten eines Übersetzerlaufs unterschiedliche Implementierungen einer Funktion angewendet werden, ohne daß sich die Identität der Objekte ändert.

Um diese Problematik, die eine objektorientierte Implementierung ausschließt, zu verdeutlichen, wird nochmals der Übersetzer aus Abbildung 4.8 betrachtet. Es wird angenommen, daß Typen auf objektorientierte Art und Weise durch Unterklassen einer abstrakten Oberklasse `Type` dargestellt werden. Eine Methode `subtype` beschreibt die Untertypbeziehung zwischen zwei Typen. Eine mögliche Implementierung, die Klassen-Typen für einen Java-Übersetzer darstellt, könnte folgendermaßen aussehen:

```
class ClassType extends Type {
   ...
   boolean subtype(Type t) {
      < Java Untertyp – Relation >
   }
   ...
}
```

Erweitert man den Java-Übersetzer beispielsweise um erweiterbare algebraische Typen, muß man die `subtype`-Methode an die neuen Untertypbeziehungen anpassen. Die Klasse `ClassType` muß erweitert werden:

```
class ExtendedClassType extends ClassType {
   ...
   boolean subtype(Type t) {
      < Erweiterte Java Untertyp – Relation >
   }
   ...
}
```

Im erweiterten Übersetzer werden Klassen-Typen durch `ExtendedClassType`-Objekte repräsentiert. Der erweiterte Compiler übersetzt in der Transformationsphase (siehe Abbildung 4.8) die erweiterten algebraischen Typen in reguläres Java. Der transformierte

Strukturbaum wird dann in der anschließenden semantischen Analyse erneut attributiert. Diesmal jedoch mit der ursprünglichen Java-Semantik. Konsequenterweise muß hier auf die alte `subtype`-Methode zurückgegriffen werden. Das ist jedoch nur möglich, wenn ausnahmslos alle existierenden `ExtendedClassType`-Objekte in `ClassType`-Objekte konvertiert werden. Dies ist technisch praktisch unmöglich, da hierfür alle Referenzen auf `ExtendedClassType`-Objekte bekannt sein und entsprechend konsistent geändert werden müssen.

Objektorientiert löst man dieses Problem gewöhnlich mit einer *Bridge* [G+95]. Es ist jedoch nicht sinnvoll, für jedes Objekt ein eigenes Implementationsobjekt zu erzeugen, auf das die Methoden umgelenkt werden. Dies würde beim Austauschen der Operationen dazu führen, daß alle Implementationsobjekte ausgetauscht werden müßten. Die einzige sinnvolle Lösung besteht also darin, für einen Datentyp genau ein „Implementationsobjekt" zu instantiieren, das Funktionen auf dem Typ anbietet. Die am Anfang des Abschnitts erläuterte Trennung zwischen Datentyp, AbstractFactory und Funktions-Modul basiert genau auf dieser Erkenntnis. Aufgrund der Ergebnisse von Kapitel 2 werden uniform stets algebraische Typen verwendet. Damit wird sichergestellt, daß jederzeit neue Varianten und neue Funktionen zu einem Typ hinzugefügt werden können. Das Context-Component-Muster stellt seinerseits sicher, daß es möglich ist, Funktionen später beliebig zu modifizieren bzw. die Funktions-Module zu erweitern.

4.4.6 Erweiterung eines Übersetzers

Ein Übersetzer der gemäß der Architekturrichtlinien der vorangehenden Abschnitte entworfen wird, läßt sich mit folgenden vier Schritten erweitern:

1. Algebraische Typen sind um neue Varianten zu ergänzen bzw. bestehende Varianten sind zu erweitern. Als Konsequenz müssen die Factory-Klassen erweitert werden, so daß anstelle der alten Typen die neuen verwendet werden.

2. Bestehende Komponenten sind durch Unterklassenbildung zu erweitern, um

   (a) das Verhalten existierender Methoden durch Überschreiben zu verändern und

   (b) die Funktionalität einer Komponente durch neue Methoden zu erweitern.

3. Neue Komponenten sind zu implementieren.

4. Eine erweiterte Kontext-Hierarchie ist aufzubauen, in der alle neuen Komponenten anstelle der alten eingesetzt werden. Damit wird der neue Übersetzer letztendlich konfiguriert.

Man baut einen erweiterten Übersetzer also dadurch, daß man die Unterschiede in Form von Unterklassen implementiert und ansonsten den alten Übersetzer wiederverwendet.

Der alte Übersetzer existiert als solches noch, da keine Codemodifikationen notwendig sind. Man bezeichnet diese Form der Programmentwicklung auch als *programming by difference*[4] [RI98]. Kapitel 4.5 stellt einen erweiterbaren Java-Übersetzer vor, der vollständig in Java mit erweiterbaren algebraischen Typen implementiert wurde. Die späte Bindung macht es möglich, daß alle Klassendateien auch vom erweiterten Übersetzer mit verwendet werden und daß lediglich die neu implementierten Komponenten und Kontexte zu übersetzen sind.

4.5 Ein erweiterbarer Java-Übersetzer

Dieses Kapitel beschreibt den im Rahmen dieser Arbeit entwickelten frei erweiterbaren Java-Übersetzer *JaCo*. Mit der Implementierung dieses Übersetzers sollte gezeigt werden, daß die in Kapitel 4.4 beschriebenen Konzepte durchaus in einem Übersetzer für eine real existierende, gängige Programmiersprache eingesetzt werden können. Der Übersetzer wurde um erweiterbare algebraische Datentypen erweitert. Die nicht-erweiterte Version von JaCo ist bereits selbst mit erweiterbaren algebraischen Datentypen geschrieben. In diesem Abschnitt wird nur ein grober Überblick über die Struktur des Übersetzers gegeben. Details können dem Quelltext entnommen werden.

Die Implementierung von JaCo baut im Bereich der Semantischen Analyse und der Codeerzeugung auf einigen Komponenten des Pizza-Compilers [Ode97] auf. Diese wurden nach Java portiert und alle Erweiterungen die Pizza betreffen entfernt. Anschließend wurden die Komponenten so umgeschrieben, daß sie in die erweiterbare Übersetzerarchitektur homogen eingebettet werden konnten. Die Erweiterung von JaCo um erweiterbare algebraische Typen wurde vollständig neu entwickelt.[5]

Die Architektur des Übersetzers wird im folgenden mittels zweiter Sichten beschrieben: Dem *statischen Aufbau* des Systems aus Komponenten (siehe 4.4.3) und dem *dynamischen Ablauf* eines Übersetzungsvorgangs (siehe 4.4.4). Der statische Aspekt betrifft die strukturelle Gliederung des Systems und wird als Instanz des Context-Component-Musters beschrieben. Der dynamische Ablauf eines Übersetzers wird in Form einer hierarchischen Schachtelung von Übersetzerphasen dargestellt.

4.5.1 Architektur des Java-Übersetzers

Abbildung 4.10 stellt die Architektur des Java-Übersetzers als Instanz des Context-Component-Musters dar. Der initiale Kontext `JavaContext` wird mittels einer Factory-

[4] Im Prinzip ist dies eine spezielle Form von aspektorientiertem Programmieren.

[5] Da bereits die nicht-erweiterte Version von JaCo mit erweiterbaren algebraischen Typen geschrieben ist, wurde vor der Entwicklung von JaCo ein Prototyp eines Pizza-Compilers mit erweiterbaren algebraischen Typen erstellt, mit welchem JaCo anfangs übersetzt werden mußte. Erst nach der Fertigstellung der erweiterten JaCo-Version konnte der Compiler sich selbst übersetzen.

Methode eines JavaSettings-Objekts erzeugt. Dieses Objekt kapselt globale Einstellungen des Übersetzers. Es kann beispielsweise über eine Kommandozeile initialisiert werden. Mit dem JavaContext wird der Übersetzer in Form der JavaCompiler-Komponente instantiiert. Globale Module werden im dazugehörigen MainContext definiert. Die Komponente JavaCompiler ist selbst keine Phase. Sie initialisiert die Strukturbäume und wendet dann darauf den eigentlichen Java-Übersetzer, d.h. die Compiler-Phase an.[6]

Die funktionale Dekomposition eines Übersetzerlaufs, repräsentiert durch die Compiler-Phase, kann Abbildung 4.11 entnommen werden.

4.5.1.1 Basissystem

Für den Übersetzer globale Komponenten werden im MainContext aufgeführt. Es folgt eine kurze Beschreibung dieser Komponenten.

ErrorHandler Dieses Modul implementiert Routinen zur Behandlung von Fehlern. Es gibt Methoden zur Ausgabe von Fehlern, Warnungen und *Deprecation*-Mitteilungen. Die Fehlermeldungen werden aus einer ResourceBundle gelesen. Damit läßt sich der Übersetzer leicht lokalisieren.

Mangler Hier wird das für die Übersetzung von inneren Klassen benötigte *Name Mangling* implementiert. Es werden Routinen zur Verfügung gestellt, die logische Klassennamen auf „physikalische“ abbilden.

PrettyPrinter Strukturbäume lassen sich mit dem PrettyPrinter lesbar auf dem Bildschirm oder in eine Datei ausgeben. PrettyPrinter ist eine Phase die lediglich Seiteneffekte erzeugt.

Disassembler Der Disassembler ist ebenso wie der PrettyPrinter eine Phase, die den Strukturbaum unverändert läßt. Er ist nur nach der Codeerzeugung sinnvoll einsetzbar. Der erzeugte Bytecode wird lesbar ausgegeben.

Classfiles Verwaltet den Zugriff auf den Klassenpfad.

[6]Es stellt sich hier die Frage, wieso die Komponente JavaCompiler und die Phase Compiler nicht kombiniert wurden. Der Grund hierfür liegt darin begründet, daß Java-Übersetzer neben den angegebenen Quellen auch Quelltexte für die Klassen übersetzen, von denen die Klassen in den angegebenen Quellen abhängen, für die aber keine Klassendateien vorliegen. Bereits für nicht erweiterbare Übersetzer bereitet diese konzeptionelle Rückkopplung zwischen der semantischen Analyse, in der festgestellt wird, daß eine weitere Quelldatei zu übersetzen ist, und der lexikalischen Analyse große Probleme. Für erweiterbare Übersetzer ist die Situation noch wesentlich schlimmer, da hier jederzeit – also nicht nur innerhalb der semantischen Analyse – dynamisch neue Quellen hinzukommen können. Das Context-Component-Muster erlaubt glücklicherweise jedoch eine relativ einfache Lösung. Neue Quellen werden nur soweit bearbeitet, daß ihre Deklarationen in den Definitionstabellen eingetragen sind. Ansonsten wird ihre Übersetzung verzögert, bis die ursprünglich zu übersetzenden Quellen vollständig bearbeitet sind. Die neuen Quellen werden dann anschließend in einem neuen Compilerlauf übersetzt. Es ist also wichtig, daß die Compiler-Phase mehrmals instantiiert werden kann.

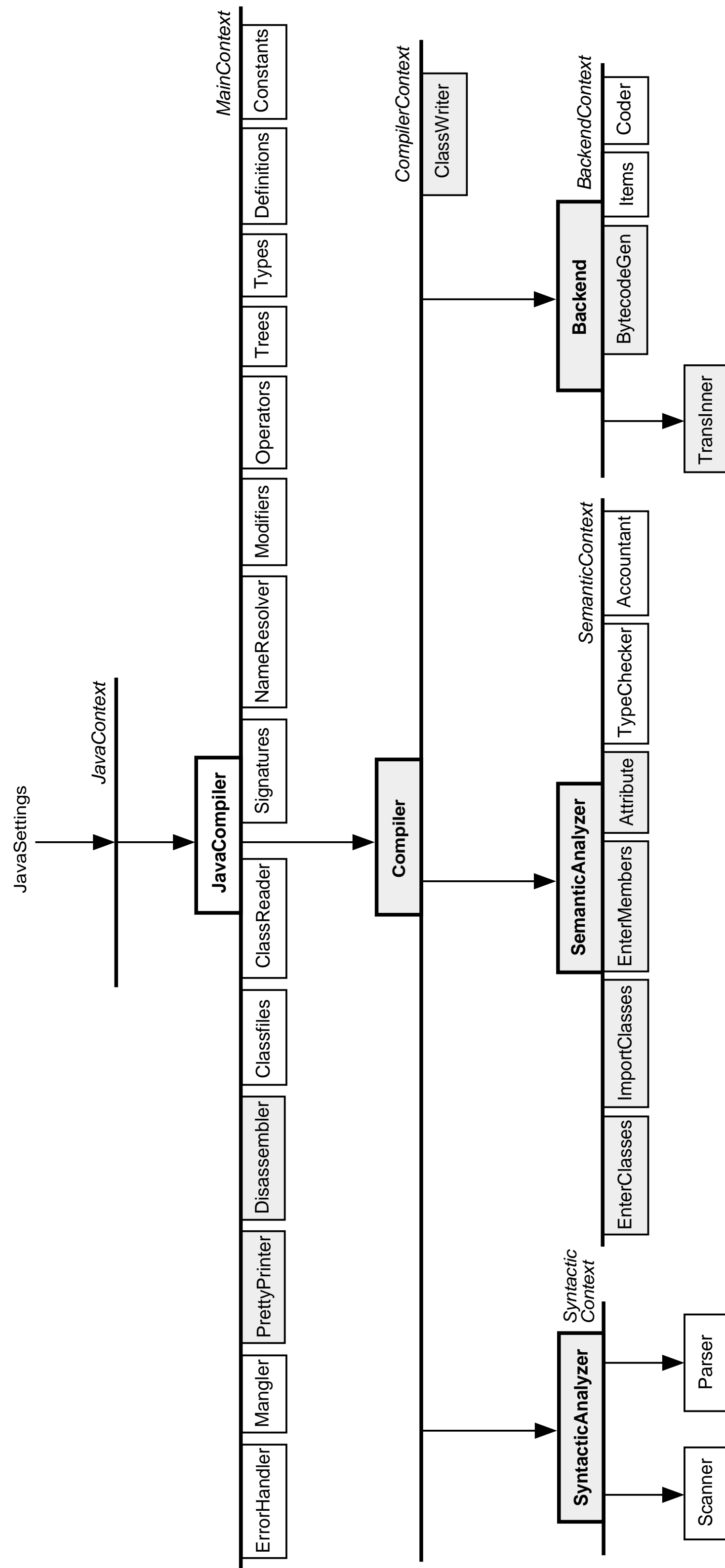


Abbildung 4.10: Architektur eines erweiterbaren Java-Übersetzers

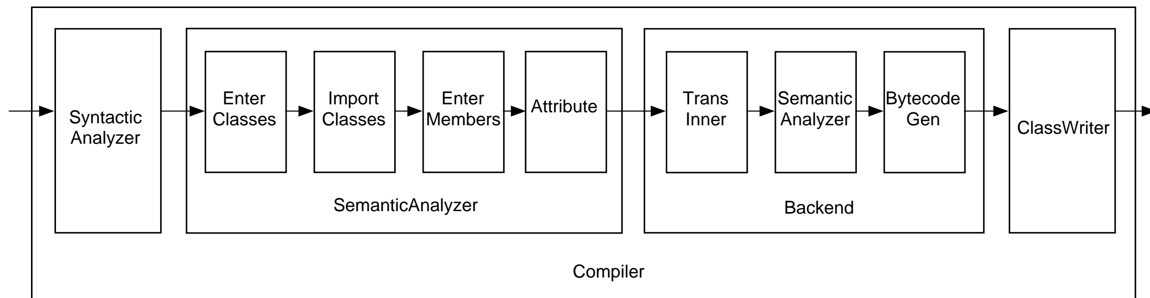


Abbildung 4.11: Phasen des erweiterbaren Java-Übersetzers

ClassReader Dieses Modul bietet Routinen zum Laden von Klassendateien. Da das Klassendateiformat festgelegt ist, dürften Erweiterungen dieser Komponente eher selten notwendig werden. Erweiterungen erlaubt das Klassendateiformat selbst nur in Form von neuen Attributen. Attribute werden nicht direkt vom ClassReader geladen, sondern es wird hierzu auf ein `AttributeReader`-Objekt zugegriffen. Über dieses Objekt können neue Attribute auf äußerst einfache Art und Weise definiert werden.

Signatures Über dieses Modul werden Typen in Typsignaturen und umgekehrt konvertiert. Typsignaturen repräsentieren Typen in Klassendateien. Dieses Modul bietet eine weitere Möglichkeit zur Erweiterung des Klassendateiformats.

NameResolver Die NameResolver-Komponente bietet verschiedene Methoden zur Namensauflösung. Die Komponente ist dafür zuständig, für einen Bezeichnernamen, die in einem Gültigkeitsbereich zugehörige Definition zu finden.

Modifiers Das Modifiers-Modul definiert Methoden zum Umgang mit Modifikatoren der Quellsprache.

Operators Dieses Modul vereinbart vordefinierte Operatoren von Java und legt Operatorbezeichnungen und Vorrangstufen fest.

Trees Mit der Trees-Komponente werden Operationen auf Strukturbäumen zur Verfügung gestellt. Außerdem wird hier die AbstractFactory für die einzelnen Konstrukte der abstrakten Syntax vereinbart. Es gibt Methoden, die Strukturbäume für komplexere Sprachkonstrukte generieren, Strukturbäume kopieren usw.

Types Die Types-Komponente bietet Operationen auf Typen und Typmengen. Außerdem werden hier die Basistypen von Java definiert und grundlegende Java-Klassen wie z.B. `java.lang.Object` geladen. Über eine AbstractFactory lassen sich Instanzen für Typ-Objekte erzeugen.

Definitions Operationen auf Definitionen werden mit diesem Modul zur Verfügung gestellt. Mit verschiedenen Tabellen werden Packages und geladene Klassen verwaltet.

Constants Das Constants-Modul definiert Operationen auf Konstanten. Großen Raum nimmt hier die Implementierung der Konstantenfaltung für Java ein.

Compiler Diese Phase repräsentiert einen vollständigen Übersetzerlauf. Die Komponente besitzt einen lokalen Kontext `CompilerContext`, welcher vier Unterphasen definiert:

1. In der *syntaktische Analyse*-Phase werden die zu übersetzenden Quelltexte eingelesen und vom Zerteiler in einen äquivalenten Strukturbaum überführt.
2. Während der *semantischen Analyse* wird der Strukturbaum auf semantische Korrektheit hin überprüft. Dabei werden für jeden Knoten des Baumes zugehörige Attributwerte berechnet.
3. Das *Backend* übersetzt zunächst die inneren Klassen von Java 1.1 in äquivalenten Java-Code ohne innere Klassen, attributiert den modifizierten Strukturbaum anschließend und erzeugt schließlich für alle Methoden, Konstruktoren und Klasseninitialisierungsblöcke direkt Bytecode. Dieser wird als Attribut im Strukturbaum abgelegt.
4. Die *Klassendateiausgabe*-Phase traversiert alle Strukturbäume und gibt für die Klassendeklarationen jeweils eine Klassendatei aus.

4.5.1.2 Syntaktische Analyse

Für die syntaktische Analyse werden zwei unabhängig voneinander instantiierbare Komponenten *Scanner* und *Parser* definiert. Der Scanner zerteilt den eingelesenen Quelltext in eine Sequenz von *Token*, welche vom Parser in einzelne Sätze der Sprache eingeteilt werden. Die Scanner-Komponente ist von Hand implementiert und kann beispielsweise recht einfach um neue Schlüsselwörter erweitert werden. Der Parser wird aus einer LALR(1)-Grammatik generiert. Die eingesetzte LALR(1)-Grammatik für Java entspricht im großen und ganzen der Version in [GJS96]. Sie wurde noch um die neuen Eigenschaften von Java 1.1 erweitert. Die Aktionen in der Grammatikspezifikation erzeugen jeweils Teile des Strukturbaums. Für jeden Quelltext wird ein eigener Scanner und ein eigener Parser instantiiert.

Der einzige zur Zeit für Java verfügbare LALR-Parser-Generator JavaCUP [H+98] ist zwar in der Lage, Grammatiken in der Größenordnung der Java 1.1-Grammatik zu bearbeiten, kann aber aus technischen Gründen hierfür keinen verifizierbaren Bytecode erzeugen. Zudem sind mit JavaCUP generierte Parser äußerst groß, langsam und speicheraufwendig. Aus diesem Grund entstand im Rahmen dieser Arbeit eine Weiterentwicklung von JavaCUP: *jcup*. jcup besitzt eine optimierte Codeausgabe und die erzeugten Parser laufen mit einem eigenen Treiber. jcup-Parser sind im Durchschnitt um den Faktor 2.5[7] kleiner und um den Faktor 4 schneller als mit JavaCUP generierte Parser.

[7] Die Klassendateien des von JavaCUP generierten Java-Parser sind insgesamt 278 KBytes groß. Der jcup-Parser besitzt dagegen insgesamt lediglich eine Codegröße von 107 KByte.

4.5.1.3 Semantische Analyse

Die Phase der semantischen Analyse setzt sich aus vier Unterphasen zusammen. Es folgt eine Beschreibung der einzelnen Unterphasen sowie der weiteren lokalen Komponenten des `SemanticContext`. In [Zen96] wird für den Übersetzer EspressoGrinder die semantische Analyse ausführlich diskutiert. Viele darin gemachten Aussagen sind auch auf JaCo übertragbar.

EnterClasses In dieser Phase werden sämtliche Klassendeklarationen gesammelt und in den Definitionstabellen der jeweiligen Gültigkeitsbereiche eingetragen. Nach dieser Phase sind alle definierten Typen bekannt. Das Aufsammeln der Klassendeklarationen muß vorab erfolgen, weil in Java eine Klasse im Quelltext nicht vor einer Verwendung als Typangabe vereinbart werden muß.

ImportClasses Diese Phase besucht alle `import`-Anweisungen und trägt die hierin importierten Klassen in den globalen Definitionstabellen ein. Damit sind die Definitionstabellen, was die Typen betrifft, vollständig.

EnterMembers In diesem Durchlauf werden die Variablen und Methoden einer Klasse in die Definitionstabellen der lokalen Gültigkeitsbereiche eingetragen und diesbezügliche Konsistenzprüfungen durchgeführt.

Attribute In dieser Phase findet eine vollständige Attributierung der Strukturbäume statt. Dabei sind im wesentlichen vier Aufgaben zu erledigen:

1. Synthetisierung des Typs aller Konstrukte im Strukturbaum
2. Typverifizierung aufbauend auf dem ermittelten Typ
3. Namensauflösung und damit zusammenhängende Überprüfungen
4. Sprungzielermittlung und Sammlung von Kontexinformationen für Sprunganweisungen.

TypeChecker Dieses Modul bietet verschiedene Funktionen an, die vor allem zur Typverifikation eingesetzt werden.

Accountant Die Accountant-Komponente führt zum einen über den Verlauf der semantischen Analyse Buch, indem sie Zugriff auf phasenübergreifende Datenstrukturen erlaubt. Zum anderen werden Factory-Methoden für *Umgebungen* definiert. Eine Umgebung kapselt jeweils eine bestimmte Menge von Attributen [Zen96].

4.5.1.4 Backend

Das Backend von JaCo hat zwei Aufgaben: zum einen müssen innere Klassen in TopLevel-Klassen transformiert werden, zum anderen ist für jede Methode Bytecode zu erzeugen.

Dieser wird jeweils als Attribut im Strukturbaum abgespeichert. Wie man Abbildung 4.11 entnehmen kann, setzt sich die Backend-Phase aus insgesamt drei Unterphasen zusammen: `TransInner`, `SemanticAnalyzer` und `BytecodeGen`. Alle im `BackendContext` definierten Komponenten werden nun näher erläutert:

TransInner Diese einfache Phase transformiert einen Java 1.1-Strukturbaum so, daß Klassendeklarationen im allgemeinen nicht mehr geschachtelt sind. Nach dieser Transformation ist der Strukturbaum nicht mehr korrekt attributiert, weshalb sich an die TransInner-Phase noch eine semantische Analyse anschließen muß. Unter der Annahme, daß die Transformation korrekt ist, dürfen während dieser erneuten semantischen Analyse keine Fehler auftreten. Die Phase der semantischen Analyse muß nicht im `BackendContext` aufgeführt werden, da man in einem geschachtelten Kontext auch Phasen äußerer Kontexte instantiieren kann.

BytecodeGen Die BytecodeGen-Phase traversiert den Strukturbaum und erzeugt für alle Methoden Java-Bytecode [LY97]. Um bei der Codeerzeugung bereits Bytecode erzeugen zu können, der direkt in eine Klassendatei übernommen werden kann, wird parallel dazu jeweils ein Konstantenpool[8] aufgebaut. Um möglichst optimalen Code nach dem Stack-Prinzip generieren zu können, wird die Codeerzeugung verzögert. Anstatt direkt Code zu erzeugen, werden Deskriptoren, sogenannte *Items* angelegt. Diese beschreiben adressierbare Einheiten, auf die lesend und schreibend zugegriffen werden kann. Verschiedene Item-Varianten charakterisieren die verschiedenen Adressierungsarten der Zielmaschine. Wirth erläutert in [Wir96] das Prinzip allgemein. In [Zen96] wird die Codeerzeugung mit Items konkret für Java-Bytecode beschrieben.

Items Dieses Modul stellt Operationen auf Items zur Verfügung. Die Bibliothek implementiert ein vollständiges Zugriffsprotokoll. Eine AbstractFactory definiert Konstruktoren für Items.

Coder Die Coder-Komponente unterstützt die Codeerzeugung für eine bestimmte Methode. Es werden Ausgabemethoden für alle möglichen Bytecodebefehlsformate definiert. Intern verwaltet ein Code-Puffer die bereits generierten Befehle. Außerdem wird hier der methodenübergreifende Konstantenpool aufgebaut.

4.5.1.5 Klassendateiausgabe

Die Klassendateiausgabe wurde vom Backend entkoppelt, um dazwischen z.B. noch einen Bytecodeoptimierungslauf durchführen zu können. Während der `ClassWriter`-Phase wird der attributierte Strukturbaum Klasse für Klasse traversiert und in Form

[8] In einer Klassendatei werden Referenzen auf Konstanten wie z.B. Namen, Signaturen und konstante Werte stets als Index auf einen Eintrag in einem Konstantenpool angegeben. Ein Konstantenpool ist ein lineares Feld, in welchem die Werte aller in einer Klassendatei vorkommenden Konstanten zentral gesammelt werden.

von Klassendateien [LY97] in das gewünschte Klassenausgabeverzeichnis geschrieben. Zu den Merkmalen einer Klasse wird in der Klassendatei jeweils eine Menge von Attributen abgespeichert, die das Merkmal näher beschreiben. Beispielsweise besitzt jede nicht-abstrakte Methode ein Attribut `Code`, das den Bytecode der Methode enthält. Der ClassWriter kennt wie der ClassReader selbst keine Attribute, sondern delegiert das Schreiben von Attributen an eine `AttributeWriter`-Komponente. Diese Komponente kann relativ einfach um neue Attribute erweitert werden. Der ClassWriter selbst sollte also im Normalfall nicht erweitert werden müssen.

4.5.2 Erweiterung des Übersetzers

Der im vorangegangenen Abschnitt dokumentierte Java-Übersetzer JaCo wurde in Java mit erweiterbaren algebraischen Typen implementiert. Im Rahmen des Bootstrappings war es deswegen notwendig, JaCo noch um erweiterbare algebraische Typen zu erweitern, so daß der Übersetzer sich selbst übersetzen kann. In Kapitel 3.5 wurde ausführlich beschrieben, wie erweiterbare algebraische Typen in reguläres Java zu übersetzen sind. Dieser Abschnitt beschäftigt sich nun noch mit der Seite der Software-Architektur. Es wird gezeigt, wie ohne Modifikationen an bestehenden Quelltexten erweiterbare algebraische Typen in den Übersetzer aus 4.5.1 integriert wurden.

Die folgenden beiden Abbildungen stellen die Architektur des erweiterten Übersetzers wieder aus zwei Sichten dar. Abbildung 4.12 zeigt den Übersetzer als Instanz des Context-Component-Architekturmusters. Grau schattierte Kontexte und Komponenten geben Erweiterungen an. Schwarz umrandete Kontexte und Komponenten werden vom ursprünglichen Übersetzer definiert. Man kann gut erkennen, daß zur Erweiterung kaum neue Komponenten notwendig waren. Alte mußten lediglich angepaßt werden und es war notwendig, eine erweiterte Kontext-Hierarchie aufzubauen, die sicherstellt, daß anstelle der alten Komponenten die neuen verwendet werden.

Interessant ist vor allem der neue `CompilerContext`. Er definiert nun zwei unterschiedliche semantische Analysen. Die eine semantische Analyse-Phase wird ohne Veränderungen aus dem ursprünglichen Übersetzer übernommen. Die zweite Phase stellt eine Erweiterung der ursprünglichen Phase dar. Auch hier wird zwar auf die alten Komponenten zurückgegriffen, allerdings nur indirekt über die erweiterten Komponenten die algebraische Typen unterstützen. An dieser Stelle wird die Tatsache ausgenutzt, daß Erweiterungen ohne Modifikationen an bestehenden Quelltexten vorgenommen werden. Man hat zwei verschiedene semantische Analysen, eine für reguläres Java 1.1, die andere für das erweiterte Java.[9] Abbildung 4.13 zeigt die funktionale Dekomposition der Phasen. Hier kann man erkennen, wann welche der beiden semantischen Analyse-Phasen aufgerufen wird.

[9] Bei einer Erweiterung durch Quelltextmodifikationen würde man diese Situation über eine einzige semantische Analyse-Phase implementieren, bei der man über einen Schalter die Erweiterung ausblenden kann. Diese eine Phase hätte also zwei verschiedene Modi. Erweiterungen auf diesem Weg sind jedoch meist kompliziert und fehleranfällig.

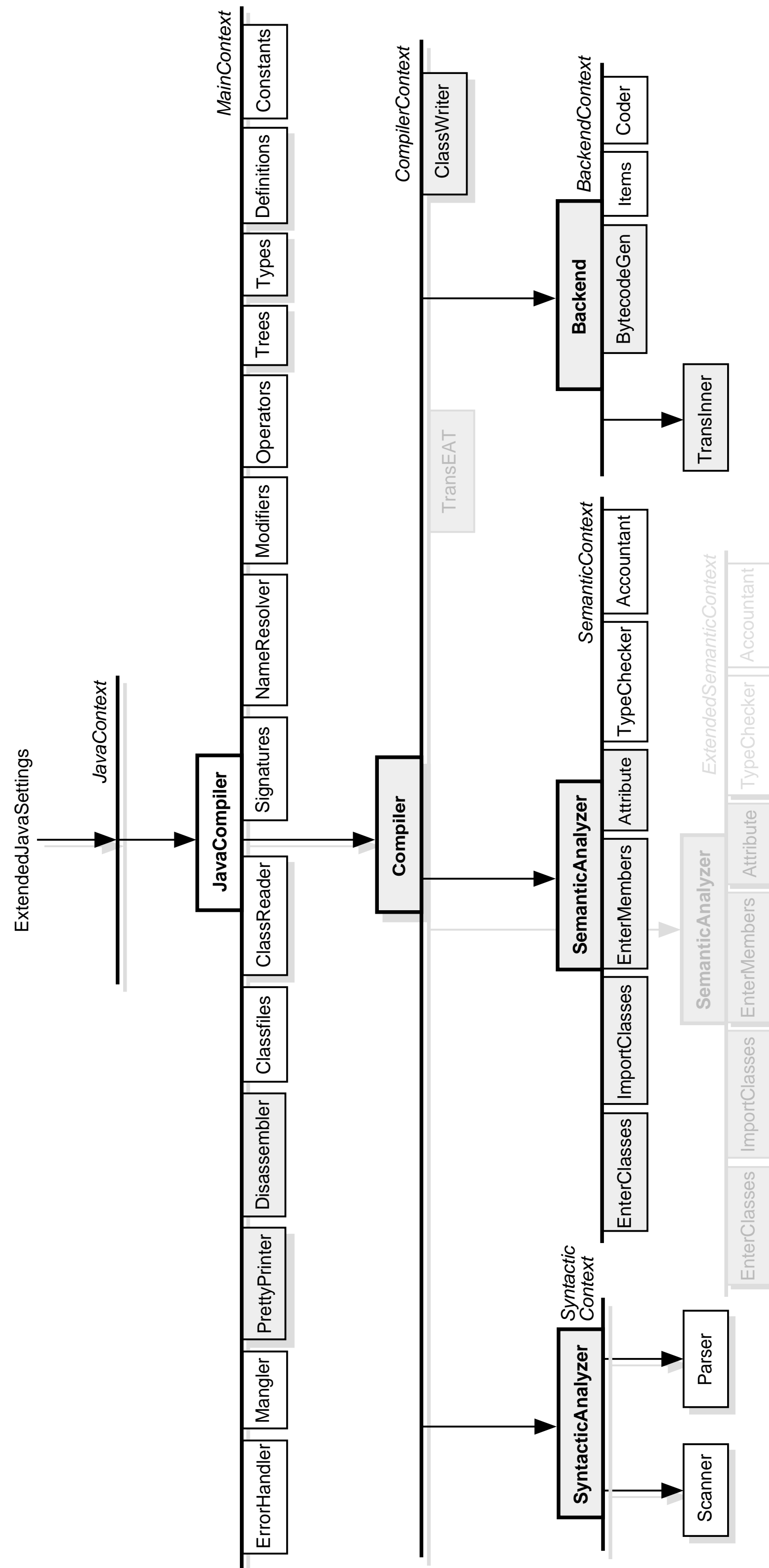


Abbildung 4.12: Architektur eines erweiterten Java-Übersetzers

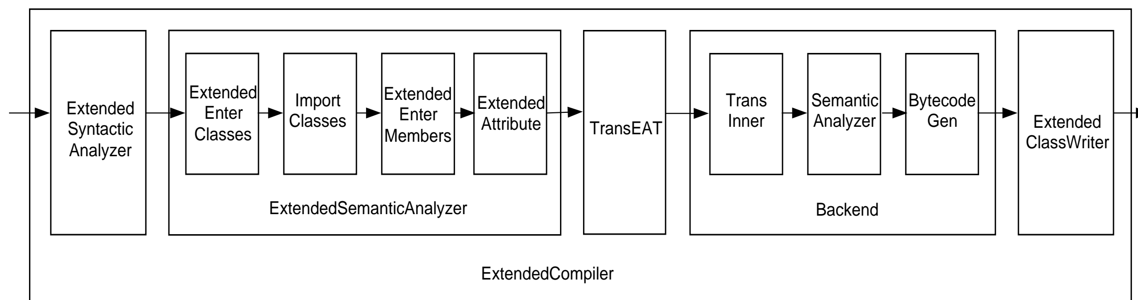


Abbildung 4.13: Phasen des erweiterten Java-Übersetzers

Für die Phase der syntaktischen Analyse mußten lediglich erweiterte Versionen für den Scanner und den Parser erstellt werden. Die Parser-Komponente wurde aus einer erweiterten Java-Grammatik mit jcup erzeugt.[10]

In der erweiterten semantischen Analyse stecken die meisten Erweiterungen, die für die Unterstützung algebraischer Typen notwendig sind. Nach der Durchführung dieser Phase wird eine neu geschriebene Übersetzungsphase `TransEAT` angewendet. Diese führt die in Kapitel 3.5 beschriebene Übersetzung des erweiterten Javas in reguläres Java durch. Die resultierenden Strukturbäume müssen nun noch mit der alten semantischen Analyse für Java 1.1 neu attributiert werden. Ist die Transformation korrekt, so werden hier niemals Fehler gefunden.

Am Backend sind keine Modifikationen notwendig, da diese Phase auf reguläre Java-Strukturbäume angewendet wird. Lediglich die Klassendateiausgabe muß ein neues Attribut unterstützen, in welchem die Typinformationen gespeichert werden, die beim Transformieren algebraischer Typen verloren gehen.

[10] Hier wurde die Java-Grammatik dupliziert und in der Kopie die nötigen Ergänzungen vorgenommen. Hätte man einen Parser-Generator der *Grammar Inheritance* unterstützt, könnte man die neue Grammatik auch als Erweiterung der alten formulieren.

Kapitel 5

Zusammenfassung

Übersetzer sind gewöhnlich komplexe Systeme, die sehr schwer zu warten und zu erweitern sind. Bei Übersetzern für eine Familie von verwandten Programmiersprachen ist die Erweiterbarkeit und Wiederverwendbarkeit von Datenstrukturen und Übersetzerkomponenten besonders wichtig. Ansonsten wäre es nur sehr schwer möglich, diese gemeinsam zu unterhalten. Bisher versuchte man dieses Problem stets durch geeignete Werkzeuge in den Griff zu bekommen, die es ermöglichen sollen, Komponenten oder Spezifikationen von Übersetzern wiederzuverwenden. Ziel dieser Arbeit war es, Konzepte zu entwickeln, mit denen erweiterbare Übersetzer unabhängig von solchen Werkzeugen implementiert werden können. Bei der Darstellung dieser Konzepte wurde Wert auf eine möglichst allgemeine Formulierung gelegt, so daß es möglich ist, diese auch beim Entwurf symbolverarbeitender Systeme im allgemeinen einzusetzen.

5.1 Beiträge der Arbeit

Die Architektur von Übersetzern folgt meist dem Architekturstil eines batch-sequentiellen Repositories. Der Übersetzungsvorgang wird hier durch eine Folge von Phasen beschrieben, die jeweils auf der internen Programmrepräsentation in Form eines abstrakten Syntaxbaums operieren. In Kapitel 2 werden verschiedene Ansätze untersucht, wie Strukturbäume und Phasen erweiterbar implementiert werden können. Es zeigt sich, daß bestehende Entwurfsstrategien dieses Problem nur sehr unzureichend lösen. Sowohl beim objektorientierten Ansatz in Form des *Interpreter*-Entwurfsmusters, als auch bei der Lösung mittels *Visitors* ergab sich, daß es entweder recht einfach ist, die Repräsentation des Syntaxbaums oder aber die Menge der Phasen zu erweitern. Der pragmatische Ansatz über *Typschalter* erlaubt zwar eine zugleiche Erweiterbarkeit von Programmrepräsentation und Phasen, kann allerdings nur recht umständlich unter Umgehung des Typsystems implementiert werden.

Aus diesem Grund führt Kapitel 3 eine erweiterbare Form von algebraischen Datentypen ein, bei denen es zugleich möglich ist, Datentypen zu erweitern, bestehende Operationen

zu verändern und neue Operationen auf einem Datentyp zu schreiben. *Erweiterbare algebraische Typen* erweisen sich damit als ideale Datentypen zur Implementierung erweiterbarer Strukturbäume und Phasen. Kapitel 3 zeigt, daß sich erweiterbare algebraische Datentypen komplikationslos in die Programmiersprache Java integrieren lassen. Neben der genauen Spezifikation dieser Typen für Java, wird ein Übersetzungsschema angegeben, mit welchem die Typen in reguläres Java übersetzt werden können. Großen Raum nimmt hier die Übersetzung von *Pattern Matching*-Ausdrücken ein. Es wird ein, zu den bisher veröffentlichten Verfahren alternatives Übersetzungsprinzip beschrieben, welches sich besonders gut zur Implementierung in imperativen Sprachen eignet. Das Verfahren hat zudem den Vorteil, daß sich damit relativ einfach prüfen läßt, ob Fallunterscheidungen vollständig sind.

Auch wenn erweiterbare algebraische Typen es auf einfache Art und Weise erlauben, erweiterbare Strukturbäume und Übersetzerphasen zu realisieren, ist letztendlich die Software-Architektur entscheided dafür, wie flexibel das gesamte System erweitert bzw. Subsysteme wiederverwendet werden können. Aus diesem Grund wird ein allgemeines Software-Architekturmuster *Context-Component* entworfen, mit welchem frei erweiterbare, hierarchisch aufgebaute Komponentensysteme konstruiert werden können. Dieses Architekturmusters propagiert eine konsequente Trennung zwischen der Komposition eines Systems und der Implementierung der einzelnen Komponenten. Dem Architekturstil eines batch-sequentiellen Mehr-Phasen-Übersetzers folgend, wird eine Variante dieses Architekturmusters für eine frei erweiterbare Übersetzerarchitektur angegeben, welche vorwiegend auf einer rekursiven Dekomposition der Übersetzerphasen basiert. Es zeigt sich, daß erweiterbare algebraische Typen und das Context-Component-Muster sich gut ergänzen: Erweiterbare algebraische Typen ermöglichen es, Typen und Funktionen flexibel zu erweitern, wohingegen das Context-Component-Muster einen Mechanismus beschreibt, zur Komposition und Erweiterung von Modulen, die Funktionen auf solchen Datentypen anbieten.

5.2 Praktische Arbeiten

Im Rahmen dieser Diplomarbeit wurde ein frei erweiterbarer Java-Übersetzer geschrieben[1], um zu zeigen, daß die entwickelten Konzepte durchaus in einem Übersetzer für eine gängige Programmiersprache eingesetzt werden können. Der Übersetzer wurde um erweiterbare algebraische Typen erweitert, so daß er in der Lage ist, sich selbst zu übersetzen. Nebenbei entstand auch ein LALR(1)-Parser-Generator basierend auf JavaCUP [H+98], da es zu dieser Zeit noch keinen LALR-Parser-Generator für Java gab, der aus einer Java 1.1-Grammatik einen halbwegs effizienten und verifizierbaren Zerteiler generieren konnte.

[1] Der Übersetzer entstand nicht vollständig neu, sondern es wurden einige Komponenten des Pizza-Übersetzers [Ode97] in adaptierter Form eingesetzt.

5.3 Ausblick

Erweiterbare algebraische Typen wurden zwar sprachunabhängig eingeführt, es wurde aber lediglich auf deren Integration in Java näher eingegangen. Es bleibt noch zu untersuchen, wie gut sich diese Typen in andere Sprachen einbettet lassen. Interessant sind hier insbesondere die Sprachen, die bereits algebraische Typen besitzen. Im Hinblick auf die Software-Architektur von Übersetzern wurde stets von einem einfachen batch-sequentiellen Modell ausgegangen. Für Fälle, in denen der Übersetzungsvorgang nicht in eine sequentielle Folge von Phasen gegliedert werden kann, lassen sich die Ergebnisse nicht unmittelbar anwenden. In konkret diesen Fällen müßte untersucht werden, inwieweit die vorgeschlagene Software-Architektur modifiziert werden kann, so daß auch diese Übersetzer frei erweitert werden können. Bei den in dieser Arbeit gemachten Untersuchungen wurden Werkzeuge weitestgehend ausgeklammert. Die vorgestellten Konzepte sollten so einfach sein, daß sie von Hand implementierbar sind. Dennoch wäre es interessant festzustellen, wie spezielle Werkzeuge den Prozeß, einen erweiterbaren Übersetzer zu entwickeln, verbessern bzw. noch weiter vereinfachen könnten.

Literaturverzeichnis

[ASU92] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau*. Addison-Wesley, 1992.

[Aßm97a] Uwe Aßmann. Software Cocktail Bars – A Model For Extensible Software. In *Proceedings of Foundations of Component Systems, Workshop of ESEC*. Zürich, 1997.

[Aßm97b] Uwe Aßmann. Meta-programming Composers in Second-Generation Component Systems. *Technical Report 17/97*, Universität Karlsruhe, 1997.

[Aug85] Lennart Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*. pages 368-381, 1985.

[BLR96] Gerald Baumgartner, Konstantin Läufer and Vincent F. Russo. On the Interaction of Object-Oriented Design Patterns and Programming Languages. *Technical Report CSD-TR-96-020*. Purdue University, February 1996.

[Bos96a] Jan Bosch. Delegating Compiler Objects – An Object-Oriented Approach to Crafting Compilers. In *Proceedings of Compiler Construction '96*, pages 326-340, 1996.

[Bos96b] Jan Bosch. Delegating Compiler Objects: Modularity and Reusability in Language Engineering. University of Karlskrona/Ronneby, 1996.

[Bos96c] Jan Bosch. Design Patterns as Language Constructs. University of Karlskrona/Ronneby, 1996.

[Bos97] Jan Bosch. Compiler Support for Extensible Languages. University of Karlskrona/Ronneby, 1997.

[BC97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*. Atlanta, 1997.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*. October 1992.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets Inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer Languages*. pages 282-290, Washington, DC, April 1992.

[B+98] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ Specification. May, 1998.

[B+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*. 17(4):471-522, December 1985.

[Car97] Luca Cardelli. *Type Systems*. Digital Equipment Corporation, Systems Research Center, 1997.

[CC96] Jian Chen and Iwan Tjuwito Chau. The Hierarchical Dependence Diagram: Improving Design for Reuse in Object-Oriented Software Development. Monash University, 1996.

[DS96] Dominic Duggan and Constantin Sourelis. Mixin Modules. In *ACM SIGPLAN International Conference on Functional Programming*. pages 262-273, May 1996.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988. ISBN 0-201-19249-7.

[Fin95] Robert B. Findler. Modular abstract interpreters. Unpublished manuscript. Carnegie Mellon University, June 1995.

[FKF98] Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen. Classes and Mixins. In *Symposium on Principles of Programming Languages*. San Diego, CA, January 1998.

[GH98] Etienne Gagnon and Laurie Hendren. SableCC - An Object-Oriented Compiler Framework. Unpublished manuscript. School of Computer Science, McGill University, Quebec, March 1998.

[Gag98] Etienne Gagnon. SableCC - An Object-Oriented Compiler Framework. Masters thesis. McGill University, Quebec, March 1998.

[G+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995. ISBN 0-201-63361-2.

[GJS96] James Gosling, Bill Joy and Guy Steele. *The Java™ Language Specification.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

[Gar95] David Garlan. What is Style? In *Proceedings of Dagstuhl Workshop on Software Architecture.* February 1995.

[GAO95] David Garlan, Robert Allen and John Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering.* Seattle WA, April 1995.

[GP95] David Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture. Draft, 1995.

[GS94] David Garlan and Mary Shaw. An Introduction to Software Architecture. *Technical Report CMU-CS-94-166.* Carnegie Mellon University, January 1994.

[H+98] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang and Andrew W. Appel. JavaCUP User's Manual. March 1998.
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[IR97] Yuuji Ichisugi and Yves Roudier. The Extensible Java Preprocessor Kit and a Tiny Data-Parallel Java. Electrotechnical Laboratory, University of Tsukuba, Japan.

[Jay95] C. Barry Jay. A Semantics for Shape. In *Science of Computer Programming,* 25:251-283, 1995.

[KFF98] Shriram Krishnamurthi, Matthias Felleisen and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. *Technical Report TR98-299.* Rice University, January 1998.

[LHJ92] Sheng Liang, Paul Hudak and Mark Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages.* pages 333-343, 1992.

[LY97] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification.* Java Series, Sun Microsystems, 1997. ISBN 0-201-63452-X.

[Mar96] Robert C. Martin. Acyclic Visitor. In *Proceedings of the Third Annual Conference on Pattern Languages of Programs,* 1996.

[MTH90] Robin Milner, Mads Tofte and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[M+97] Robert T. Monroe, Drew Kompanek, Ralph Melton and David Garlan. Stylized Architecure, Design Patterns and Objects. In *IEEE Software*. pages 43-52, January 1997.

[Nor96] Martin E. Nordberg. Variations on the Visitor Pattern. Quintessoft Engineering, Inc, July 1996.

[Ode97] Martin Odersky. Pizza Distribution, University of South Australia, 1997. `http://www.cis.unisa.edu.au/~pizza`.

[OP95] Martin Odersky and Michael Philippsen. EspressoGrinder Distribution. University of Karlsruhe, 1995. `http://wwwipd.ira.uka.de/~espresso`.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*. pages 146-159, 1997.

[PJ97] Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. Technical Report 05, University of Technology, Sydney, 1997.

[PJN98] Jens Palsberg, C. Barry Jay and James Noble. Experiments with Generic Visitors. Unpublished Manuscript, 1998.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. In *ACM SIGSOFT Software Engineering Notes*. 17(4):40-52, October 1992.

[Pet97] John Peterson et.al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*. Version 1.4, January 1997.

[PJ86] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, May 1986.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty – transparent remote objects in Java. In *Concurrency: Practice and experience*. 9(11):1225-1242, November 1997.

[EC96] The E Extension to Java – A white paper. Electric Communities, Cuppertino CA. `http://www.communities.com/products/tools/e/`.

[RI98] Yves Roudier and Yuuji Ichisugi. Mixin Composition Strategies for the Modular Implementation of Aspect Weaving – The EPP preprocessor and its module description language Ld-2. *Aspect Oriented Programming Workshop at ICSE*. Kyoto, April 1998.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SNH95] D. Soni, R. Nord and C. Hofmeister. Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering*. pages 196-207, Seattle WA, April 1995.

[TC97] Michiaki Tatsubori and Shigeru Chiba. Open Java 1.0 API and Specification. Draft 1.0a, University of Tsukuba, October 1997.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, Berlin, 1984. ISBN 3-540-90821.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer-Verlag, 1997. ISBN 3-540-61692-6.

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996. ISBN 3-89319-931-4.

[Zen96] Matthias Zenger. Architektur von EspressoGrinder. In Java Seminarbeiträge, *Technical Report 24/96*. Universität Karlsruhe, 1996.

[Zen97] Matthias Zenger. Transparente Objektverteilung in Java. Studienarbeit, Universität Karlsruhe, 1997.