Java Bytecode Generierung von überladenen Methoden

Studienarbeit

des Studiengangs Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart-Horb

von

Enrico Schrödter

5. Juni 2016

Bearbeitungszeitraum	5 und 6 Semester
Matrikelnummer, Kurs	3209114, TINF2013
Betreuer	Prof. Dr. rer. nat. Martin Plümicke
	Herr Andreas Stadelmeier

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

„Java Bytecode Generierung von überladenen Methoden“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Horb, 5. Juni 2016

Abstract

In der Programmiersprache Java werden Typangaben vom Entwickler benötigt. Diese können jedoch auch von einem Computer berechnet werden. Mit diesem Thema beschäftigt sich das Java-TX Projekt, welches die Basis für diese Studienarbeit ist. In dieser Studienarbeit wurde die Generierung des Bytecodes überarbeitet und überladene Methoden ermöglicht. Der Fokus lag dabei auf den generischen Typen, die durch Type Erasure ein besonderes Problem bereiten. Die Unterscheidung von generischen Datentypen wurde umgesetzte, indem für jede Variation eine eigene Klasse generiert wird.

Danksagungen

Einen besonderen Dank gilt Prof. Dr. Plümicke, der diese Studienarbeit ermöglicht hat, da er sein Forschungsthema in einem Forschungsprojekt von Studenten unterstützt umsetzt. Er liefert theoretische Konzepte, die von Studenten aufbereitet und praktisch umgesetzt werden. Diese Art von Forschung hat mich einen großen Schritt weitergebracht.

Ebenso danke ich Herr Stadelmeier für die wöchentlichen Treffen, in denen wir gemeinsam Probleme erörtert und gelöst haben. Er hält den Überblick über das ganze Projekt und konnte mich unterstützen, sodass ich mich in der über Jahre angewachsene Codebasis relativ schnell zurechtfinden konnte.

Inhaltsverzeichnis

Abkürzungsverzeichnis

JVM	Java Virtual Machine
AST	abstrakten Syntaxbaum
JDK	Java Development Kit
JRE	Java Runtime Environment
BCEL	Byte Code Engineering Library

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Seit nunmehr über 10 Jahren wird an der DHBW Stuttgart am Campus Horb ein Java-Compiler entwickelt, der keine Angaben von Typen benötigt. Die Typen werden durch einen Algorithmus automatisch berechnet. Mit dieser Arbeit, soll dieser Compiler erweitert werden, sodass die volle Stärke der Typenberechnung zu tragen kommen kann.

Bei der Berechnung der Typen werden mehrere Typisierungen generiert. Aktuell wird jeweils eine Lösung verwendet um eine Klasse zu generieren. In der Erweiterung sollen möglichst viele Lösungen für die Generierung von Bytecode hinzugezogen werden, sodass die generierten Klassen noch allgemeiner werden. Sind die Typen der Parameter einer Methode nicht eindeutig, so soll diese überladen werden. Dabei ist die interne Repräsentation von Variablen in der Java Virtual Machine (JVM) zu beachten. Diese muss soweit abgeändert werden, dass es möglich ist alle, aber im besonderen die generischen Typen von Java zu verwenden.

Der aktuelle Compiler kann Standard-Java ohne Typen parsen und dieses mithilfe eines Typinferenzalgorithmus typisieren. Die Typinferenz wird parallel zu dieser Arbeit überarbeitet, indem die Typunifikation neue entwickelt wird [Ste16]. Die Bytecodegenerierung, die der Schwerpunkt dieser Arbeit ist, kann die wichtigsten Elemente von Java in Bytecode übersetzen [Fik+15]. Die restlichen Elemente sind für ein Forschungsprojekt, das sich mit Typisierung beschäftigt, nur nebensächlich.

Dies Erweiterung des Compilers wird iterativ umgesetzt. Dies spiegelt sich auch in dieser Arbeit wieder. Nachdem die Grundlagen und die Problemstellung erläutert wurde, wird die Umsetzung beschrieben. Diese beginnt mit einer einfachen Umsetzung, die für jede Lösung der Typinferenz eine Methode im Bytecode erzeugt. Dadurch werden Methoden überladen. Dieser Ansatz wird Schritt für Schritt verbessert, sodass er am Ende weder zu viele noch zu wenig Methode im Bytecode erzeugt.

2 Grundlagen

2.1 Compiler

Ein Compiler ist ein Programm, welches eine Programmiersprache in Maschinensprache umwandelt. Der Aufbau eines Compilers ist in 2.1 dargestellt. Die einzelnen Schritte werden in den folgenden Abschnitten erläutert.

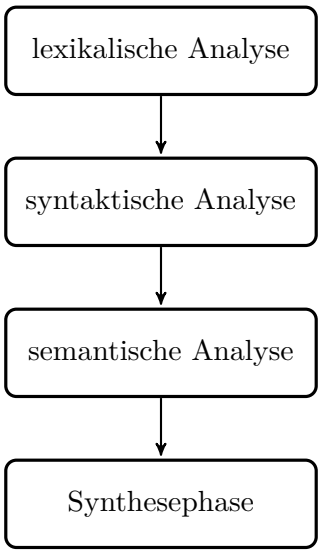

Abbildung 2.1: Aufbau eines Compilers [WSH12, S. 3]

In der **lexikalischen Analyse** wird das Programm als Zeichenkette in eine Menge von Tokens übersetzt. Ein Token ist eine lexikalische Einheit, die in den weiteren Phasen weiterverwendet wird. Die Übersetzung von Zeichen in Tokens erfolgt durch einen Scanner. Dieser gibt als Ergebnis der lexikalischen Analyse eine Menge von Tokens zurück oder bricht seine Arbeit mit einem lexikalischen Fehler ab. Ein lexikalischer Fehler tritt auf, wenn der Scanner eine Zeichenkette nicht in gültige Tokens umwandeln kann.

Nachdem der Scanner die lexikalische Analyse durchgeführt und eine Menge von Tokens zurückgegeben hat, führt der Parser die **syntaktische Analyse** auf den Tokens aus. In der syntaktischen Analyse wird der Programmcode auf die korrekte Syntax, der Grammatik der Programmiersprache, überprüft. Dafür verwendet der Parser die Ergebnismenge des Scanners und übersetzt diese in einen abstrakten Syntaxbaum (AST). Ist die Übersetzung in einen AST nicht möglich, so wirft der Parser einen syntaktischen Fehler. Dieser ist beispielsweise der Fall, wenn der Programmcode nicht der Grammatik entspricht. Beispiel für syntaktische Fehler sind falsche Klammersetzungen oder vergessende abschließenden Semikolons.

In der **semantischen Analyse** wird überprüft, ob der Programmcode semantisch korrekt ist. Die Anforderungen der semantischen Analyse werden nicht, wie bei der lexikalischen und syntaktischen Analyse durch eine Grammatik beschrieben. Die Anforderungen der semantischen Analyse ergeben sich durch das eigene und durch eingebundene Programme. Fehler die die semantische Analyse aufdeckt sind beispielsweise Typfehler oder Verstöße gegen die Sichtbarkeitsregeln. Als Resultat der semantischen Analyse erhält man einen attributierten AST.

Die letzte Phase eines Compilers ist die **Synthesephase**. In der Synthesephase wird der attributierte AST in Maschinencode übersetzt.

2.2 Java

Java ist eine Programmiersprache und ein Teil der Java-Technologie. Die Java-Technologie umfasst die Programmiersprache Java, das Java Development Kit (JDK) und die Java Runtime Environment (JRE).

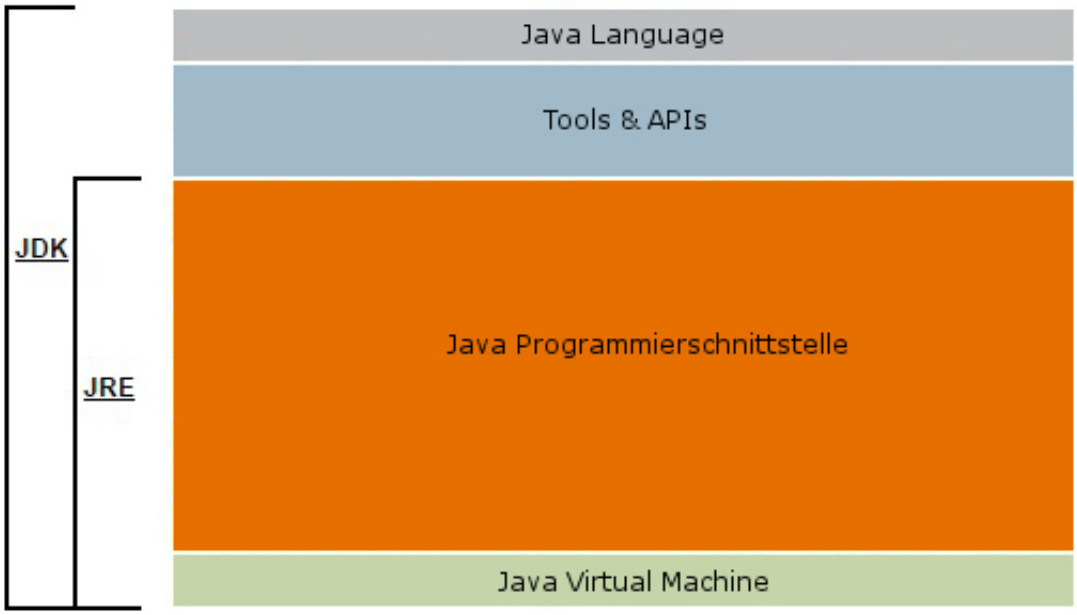

Abbildung 2.2: Die Java Technologie [Ora16]

Die Programmiersprache Java kann auf heute gängigen Betriebssystemen, wie Windows, Mac OS X oder Linux, verwendet werden. Die Entwicklung von Java ist dabei unabhängig von dem Betriebssystem. Entwickelt man auf einem Betriebssystem Java, so kann man es auf jedem anderen Betriebssystem auch verwendet. Die Betriebssysteme bilden die unterste Schicht der Java-Technologie auf der die JDK und die JRE aufbauen.

Um Java in Bytecode zu übersetzen benötigt jeder Entwickler die JDK. Diese Enthält verschiedene Entwicklungswerkzeuge und kann Java-Dateien in ausführbare Dateien, wie *Class*-Dateien oder *Jars* übersetzen. Diese enthalten Bytecode, der ist ein optimierter Zwischencode ist, der von der JVM compiliert werden kann. Der Unterschied zwischen einer *Class*-Datei und einem *Jar* ist,

dass eine *Class*-Datei eine einzelne Datei ist, die eine Klasse repräsentiert, wogegen eine *Jar* ein Archiv ist, welche mehrere *Class*-Dateien enthalten kann und damit eine ganze Library oder Anwendung repräsentiert.

Die JVM ist in der Java Runtime Environment (JRE) enthalten. Die JRE muss von jedem Nutzer von Java installiert sein, da diese die Java-Api und die JVM zur Verfügung stellt. Die JVM ist die Einheit der Java-Technologie, die den Bytecode ausführt. Dafür besitzt sie Komponenten um Klassen zu laden, Speicher zu verwalten und um Klassen auszuführen.

2.2.1 Java Typsystem

Java ist eine statisch typisierte Sprache. Es gibt verschiedene Arten von Typen in Java die im folgenden näher betrachtet werden. [Gos+15, S. 41]

Die einfachste Art von Typen sind **primitive Datentypen**. Diese sind die elementarer Bestandteile von Java. Mit diesen können Ganzzahlen, Gleitkommazahlen, logische Werte und einzelne Zeichen gespeichert werden. Beispiele dafür sind int, double, bool und char. Für primitiven Datentypen gibt es Wrapper-Klassen, um mit diesen objektorientierte Konzepte anzuwenden. Klassen sind ein Teil einer weiteren Art von Typen, den Referenztypen.

Referenztypen ist eine weiter Art von Typen. Sie werden Referenztypen genannt, da sie über ihre Speicheradresse referenziert werden. Beispiele für Referenztypen sind Klassen, Interfaces, Arrays und null. Referenztypen bestehen aus anderen Referenztypen oder primitiven Datentypen. Ein String, der in Java kein primitiver Datentyp ist, besteht beispielsweise aus mehreren Chars, die eine Zeichenkette bilden. Mit Hilfe der Referenztypen können die Konzepte des objektorientierten Paradigma angewendet werden.

Seit Java 5 gibt es zusätzlich **generische Datentypen**. Durch generische Datentypen ist es möglich ein Problem abstrakt zu lösen und durch die Angabe eines Parameters konkret zu verwenden. Ein populäres Beispiel für einen generischen Datentyp sind die Containerklassen. In Java wird beispielsweise die Klasse Vektor einmal abstrakt programmiert und kann mit jedem Referenztyp verwendet werden. Die Verwendung der Klasse Vektor wird in der Listing 2.1 dargestellt. Es wird eine Klasse Vektor programmiert, die einen Parameter E besitzt. Diese kann beispielsweise mit den Typen Integer oder String verwendet werden.

```
1 public class Vector<E> extends AbstractList<E>{
2 ...
3 }
4
5 Vector<Integer> integerVector = new Vector<>();
6 Vector<String> stringVector   = new Vector<>();
```

Listing 2.1: Beispiel mit generischem Datentyp

Zusätzlich zu den Typparametern gibt es noch *Wildcards*. Es wird zwischen Unbounded Wildcards, Upper Bounded Wildcards und Lower Bounded Wildcards unterschieden. Bei einem Unbounded Wildcard können alle Typen verwendet werden. Dieser wird durch ein *Fragezeichen* gekennzeichnet.

Die verwendbaren Typen werden beim Upper Bound und Lower Bound Wildcard mit der Klassenhierarchie eingeschränkt. Bei Upper Bound Wildcards, dargestellt durch *? extends Typ*, können nur Typen verwendet werden, die vom angegebenen Typ abgeleitet sind. Beim Lower Bound Wildcard, dargestellt durch *? super Typ*, können dagegen nur Typen verwendet werden, die in der Klassenhierarchie niedriger sind.

```
1 Vector<?>                unbound    = new Vector<>();
2 Vector<? extends Number> upperBound = new Vector<>();
3 Vector<? super Number>   lowerBound = new Vector<>();
```

Listing 2.2: Beispiel mit Wildcards

2.2.2 Bytecode

Der Bytecode wird, wie in 2.2 dargstellt, in der Java-Technologie verwendet. Er wird aus einer Java-Datei mithilfe des JDK erzeugt und in der JVM ausgeführt.

Der Bytecode einer Java Klasse steht dabei in einer Class-File. Diese hat eine vorgegebene Struktur. Diese Struktur einer disasambelten Class-Datei wird in 2.3 von Lindholm u. a. [Lin+14, S. 70] dargestellt.

```
1  ClassFile {
2      u4             magic;
3      u2             minor_version;
4      u2             major_version;
5      u2             constant_pool_count;
6      cp_info        constant_pool[constant_pool_count-1];
7      u2             access_flags;
8      u2             this_class;
9      u2             super_class;
10     u2             interfaces_count;
11     u2             interfaces[interfaces_count];
12     u2             fields_count;
13     field_info     fields[fields_count];
14     u2             methods_count;
15     method_info    methods[methods_count];
16     u2             attributes_count;
17     attribute_info attributes[attributes_count];
18 }
```

Listing 2.3: Struktur einer Class-Datei [Lin+14]

Diese Struktur wird im Folgenden beschrieben. Besondere Aufmerksamkeit erhalten dabei die Datentypen und Methoden im Bytecode, da diese von zentraler Bedeutung für diese Arbeit sind.

Eine Class-Datei ist in verschachtelten Tabelle organisiert. Im Feld *magic* wird eine Bytefolge gespeichert, die zum schnellen Ausschluss von ungültigen Class-Dateien dient. In diesem Feld muss immer die Bytefolge *0xCAFEBABE* stehen, liegt diese Folge nicht vor kann die Verarbeitung sofort abgebrochen werden, da es keine gültige Class-Datei ist.

In den nächsten zwei Felder, *minor_version* und *major_version*, wird die Version des Classfile-Formates abgelegt. Diese sind wichtig, damit die Classfile nicht von einer JVM ausgewertet wird, die eine neuere oder ältere Version des Classfile-Formates verwendet.

Der Aufbau der zwei folgenden Feldern ist typisch für Bytecode. In *constant_pool_count* wird die Größe des *Konstantenpools* festgelegt, der eine Untertabelle im nächsten Feld bildet. Dieser Aufbau wird auch für Interfaces, Felder, Methoden und Attribute verwendet.

Der *Konstantenpool* ist eine Tabelle, die alle Konstanten und Referenzen einer Klasse enthält. Vielen Einträge in einer ClassFile beziehen sich auf den Konstantenpool. Jede Konstante muss einen von 14 Typen besitzen, die in der Tabelle 2.2.2 dargestellt sind.

Type	Wert
Class	7
Fieldref	9
Methodref	10
InterfaceMethodref	11
String	8
Integer	3
Float	4
Long	5
Double	6
NameAndType	12
Utf8	1
MethodHandle	15
MethodType	16
InvokeDynamic	18

Tabelle 2.1: Konstantenpool Tags [Lin+14, S. 79]

Mit den *access_flags* wird festgelegt welche Sichtbarkeit die Klasse besitzt, ob es eine Klasse oder ein Interface ist, ob diese veränderbar ist und ob sie abstrakt ist. Die Werte der *access_flags* beziehen sich dabei nur auf die Klasse. Auf der Ebene der Felder und Methoden gibt es zusätzliche *access_flags*. Alle Werte, die für die *access_flags* möglich sind, sind in der Tabelle 2.2.2 nach der Spezifikation [Lin+14, S. 79] dargestellt.

In *this_class* und *super_class* werden die Refernzen zur aktuellen und zur Elternklasse im Konstantenpool gespeichert.

Die restlichen Felder sind weitere Untertabelle zu den implementierten Interfaces, den Felder, Methoden und Attributen. Von diesen soll der Aufbau einer Methode im folgenden näher betrachtet werden. Genauso wie eine Klasse besitzt eine Methode *access_flags*. Diese müssen mindestens

Name	Wert	Funktion
ACC_PUBLIC	0x0001	Öffentlich
ACC_FINAL	0x0010	Keine Ableitung möglich.
ACC_SUPER	0x0020	Konfiguration für invocespecial
ACC_INTERFACE	0x0200	Interface
ACC_ABSTRACT	0x0400	Abstrakte Klasse
ACC_SYNTHETIC	0x1000	Eine vom Compiler generierte Klasse
ACC_ANNOTATION	0x2000	Annotation Typ
ACC_ENUM	0x4000	Eine Enumeration

Tabelle 2.2: Namen, Werte und Funktionen der *access_flags* [Lin+14, S. 91]

die Sichtbarkeit der Methode zwischen *public*, *protected* und *private* festlegen. Zusätzliche können Flags wie beispielsweise *final* oder *synchronized* verwendet werden. Der zweite Wert den eine Methode im Bytecode enthalten muss ist der Methodennamen. Dieser wird durch einen *UTF-8*-Eintrag im Konstantenpool definiert. Zusätzlich zu den *access_flags* und dem Namen der Methode, muss die Methode noch einen Descriptor und eine Tabelle von Attributen enthalten.

```
1 method_info {
2   u2 access_flags;
3   u2 name_index;
4   u2 descriptor_index;
5   u2 attributes_count;
6   attribute_info attributes[attributes_count];
7 }
```

Listing 2.4: Aufbau einer Methode im Bytecode

Ein Descriptor stellt einen Typ einer Methode oder eines Feldes in Textform dar. Der Descriptor einer Methode enthält dabei Informationen zu den Argumenten und dem Rückgabewert. Die Typen werden in einer verkürzten Schreibweise dargestellt, die in der Tabelle 2.2.2 dargestellt ist. Der Descriptor der Methode *add* der Klasse *List* lautet *(Ljava/lang/Object;)Z*. Am Anfang stehen die Argumenttypen, in diesem Fall eine Klasse des Types Objekt, zwischen zwei Klammern. Gefolgt werden die Argumenttypen vom Rückgabewert *Z*, der für einen booleschen Wert steht.

Ein wichtiges Attribut ist die Signatur. Die Signatur ist ein Attribut, das den Type einer Klasse, eines Interfaces, einer Methode oder eines Feldes genauer als durch die Deskription definiert, wenn sie Typevariablen enthalten oder ein parametrisierter Type ist. Im Gegensatz zum Deskriptor, enthält die Signatur alle Typevariablen. Die Signatur wird von der JVM nicht für das Laden und Verlinken von Klassen verwendet, dafür wird der Deskriptor verwendet. Die Signatur ist wichtig für das Debugging, Reflexion [1] und für das Compilieren, wenn nur die *Class*-Datei von Schnittstellen zur Verfügung stehen.

[1] Reflexion (engl. Reflection) bietet dem Entwickler die Möglichkeit zur Laufzeit Klassen zu analysieren und beispielsweise Methoden dynamisch auszuführen.

FieldType	Type	Interpretation
B	Byte	8 Bit
C	Char	UTF-16 Zeichen
D	Double	64 Bit IEEE 754
F	Float	32 Bit IEEE 754
I	Integer	32 Bit Ganzzahl
J	Long	64 Bit Ganzzahl
L ClassName;	Referenz	Instanz der Klasse ClassName
S	Short	16 Bit Ganzzahl
Z	Boolean	Wahrheitswert
[	Referenz	Eine Array Dimension

Tabelle 2.3: Typen von Feldern [Lin+14, S. 77]

2.2.3 Bytecode Debuggen

Zum Debuggen und zum Verstehen von Bytecode kann der „Java Class File Disassembler“ verwendet werden. Dieser kann den Bytecode einer vorhandenen Class-Datei anzeigen. Um den Bytecode darzustellen wird das Kommando *javap* verwendet. Die Verwendung dieses Befehl wird anhand eines Beispiel in den folgenden Listings 2.5, 2.6, 2.7 erläutert.

Als Beispiel wird eine Klasse *Beispiel* verwendet, die eine Methode *hello* besitzt, welche einen konkatenierten String zurückgibt.

```
1 class Beispiel{
2   public String hello(String argument){
3     return "Hello " + argument;
4   }
5 }
```

Listing 2.5: Beispiel für javap

Um die Klasse *Beispiel* zur disassamblen wird der Befehl *javap* verwendet, der in Listing 2.6 dargestellt. Mit dem Parameter *-verbose* wird die Ausgabe vollständig angezeigt.

```
1 javap -v Beispiel.class
```

Listing 2.6: Verwendung von javap

Der disassamblete Bytecode des Beispiels wird in Listing 2.7 dargestellt. Wichtige Teile sind die Konstantenpool und die Methoden. In den Methoden wird die Signatur, die Description, die Access_flags und der Code dargestellt. Der Code ist eine Folge von Bytecodebefehlen, die sich meist auf den Konstantenpool beziehen.

Im Beispiel besitzt die Methode *hello* den Deskriptor `(Ljava/lang/String;)Ljava/lang/String;`. Die Flag `ACC_PUBLIC` zeigt an, dass die Methode öffentlich ist. Im Codeattribute kann man erkennen,

dass das Konkatenieren von String in Java durch den *StringBuilder* durchgeführt wird. Mit *new* wird ein neuer *StringBuilder* erzeugt. Das erste *invokespecial* ruft den Konstruktor auf. Danach wird mit *ldc* bzw. *aload_1* die Strings geladen und mit der Methode *append* bzw. dem Befehl *invokevirtual* der String konkateniert. Mithilfe der *toString* Methode wird aus dem *StringBuilder* ein *String* generiert der mit *areturn* zurückgegeben wird.

```
1  Classfile /Beispiel.class
2    Last modified 25.04.2016; size 454 bytes
3    MD5 checksum 13bc0f426ccd8f5be7c1166f05c6e265
4    Compiled from "Beispiel.java"
5  class Beispiel
6    minor version: 0
7    major version: 52
8    flags: ACC_SUPER
9  Constant pool:
10    #1 = Methodref          #8.#17     // java/lang/Object."<init>":()V
11    #2 = Class              #18        // java/lang/StringBuilder
12    #3 = Methodref          #2.#17     // java/lang/StringBuilder."<init>":()V
13   ...
14 {
15   public java.lang.String hello(java.lang.String);
16     descriptor: (Ljava/lang/String;)Ljava/lang/String;
17     flags: ACC_PUBLIC
18     Code:
19       stack=2, locals=2, args_size=2
20          0: new           #2          // class java/lang/StringBuilder
21          3: dup
22          4: invokespecial #3          // Method java/lang/StringBuilder."<init
                >":()V
23          7: ldc           #4          // String Hello
24          9: invokevirtual #5          // Method java/lang/StringBuilder.append:(
                Ljava/lang/String;)Ljava/lang/StringBuilder;
25         12: aload_1
26         13: invokevirtual #5          // Method java/lang/StringBuilder.append:(
               Ljava/lang/String;)Ljava/lang/StringBuilder;
27         16: invokevirtual #6          // Method java/lang/StringBuilder.toString
               :()Ljava/lang/String;
28         19: areturn
29       LineNumberTable:
30         line 3: 0
31
32   ...
33 }
34 SourceFile: "Beispiel.java"
```

Listing 2.7: Disassambletes Beispiel mit javap

2.3 Das Java-TX Projekt

Das Java-TX Projekt beschäftigt sich mit der Entwicklung eines Compilers, der Java Quellcode ohne Angabe von Datentypen in JVM-Bytecode übersetzt. Die einzelnen Teile des Projektes sind in der Abbildung 2.3 [Plü15b] dargestellt. Die verschiedenen Teile des Projektes werden im Folgenden näher erläutert.

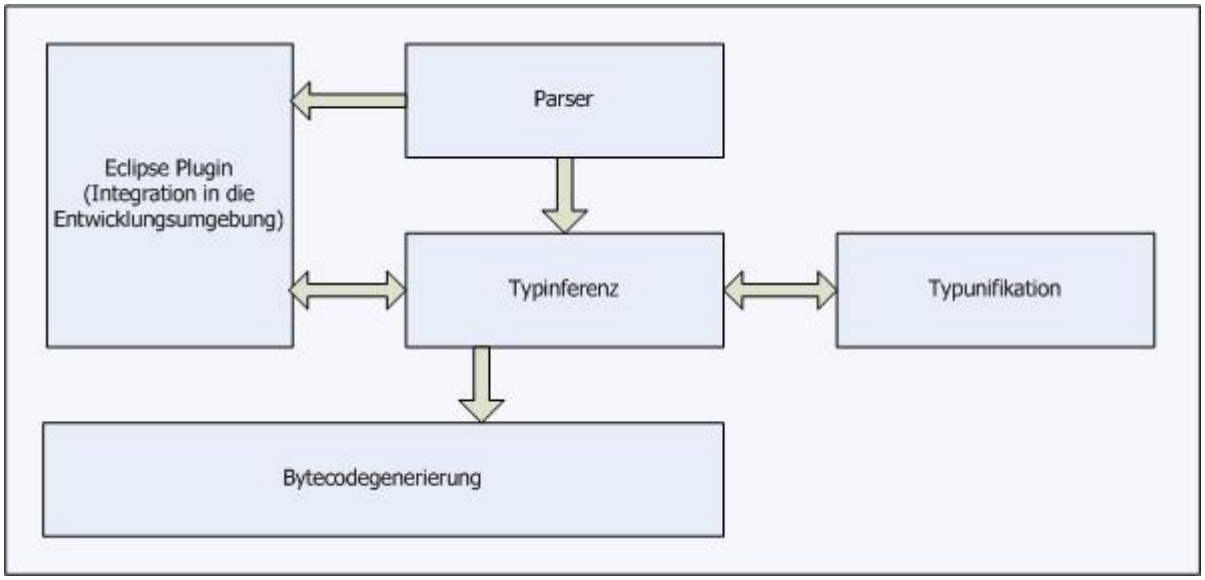

Abbildung 2.3: Java-TX Schema

Eclipse Plugin

Die Nutzerschnittstelle für das Java-TX Projekt ist ein Eclipse Plugin, welches Javacode in Bytecode umwandelt. Die Besonderheit des Javacodes ist, dass keine Typen angegeben werden müssen, da diese durch die Algorithmen bestimmt werden. Der Nutzer des Plugins muss nur Typen angeben, wenn diese nicht eindeutig bestimmt werden können. Dafür werden von dem Plugin alle möglichen Typen zur Auswahl gestellt. Diese Vorschläge können getätigt werde, da der Parser den Quellcode parset und mit Hilfe von Typinferenz alle Typen bestimmt.

Parser

Der Parser des TX-Projektes ist eine Kombination aus Scanner und Parser. Diese werden mit Hilfe von *JLex* und *jay* generiert.

JLex ist ein eine Java Anwendung, die als Generator zum Erzeugen eines Scanners dient. Der Scanner wird aus einer JLex-Spezifikation erzeugt, welche jedem Token einen regulären Ausdruck zuordnet. Der erzeugte Scanner ist wiederum eine Java Anwendung. Die Anwendung scannt einen Text und wandelt diesen in die definierten Tokens um.

Diese Tokens werden vom Parser verwendet um einen AST zu generieren. Zur Generierung wird der Parsergenerator jay verwendet, welcher eine Parserspezifikation in einen Parser übersetzt.

Typinferenz

Das Ziel der Typinferenz ist es, alle Typen zu berechnen, die nicht angegeben wurden. Dafür wird von der Typinferenz der AST verwendet um alle bekannten Typen zu erkennen. Auf diese Typen werden unter Verwendung der Typunifikation Typinferenz-Regeln angewendet. Das Ergebnis der Typinferenz ist eine Menge von gültigen Typannahmen.

Typunifikation

Die Typunifikation wird für den Typinferenzalgorithmus benötigt. Das Ziel der Unifikation ist es eine gültige Belegung der Typvariablen zu erhalten. Als Eingabe erhält diese dabei eine Menge von Typpaaren. Diese Typpaare stehen in einer Relation. Durch die Anwendung verschiedener Regeln wird aus diesen Lösungen generiert.

Bytecodegenerierung

In der letzten Komponente des Java-TX Projektes wird der AST mithilfe der Typannahmen in Bytecode übersetzt. Dieser kann durch die JVM ausgeführt werden. Die Generierung des Bytecodes erfolge dabei durch das Framework Byte Code Engineering Library (BCEL). Mithilfe von BCEL ist es möglich Bytecode zu erzeugen, zu modifizieren und zu analysieren. Der Bytecode kann erzeugt werden, indem man die gewünschte Klasse als BCEL-Objekte modelliert.[Apa14]

3 Problemstellung

Das Ziel dieser Arbeit ist es, ein Konzept zu entwickeln, sodass mit Java Methoden mit generischen Parametern, wie in Listing 3.3 dargestellt, überladen werden können. Dieses Konzept soll im Java-TX Projekt umgesetzte werden.

Bei der Berechnung der Typen durch die Typinferenz wird meist nicht nur eine Lösungen gefunden, sondern eine Menge von Lösungen. Aktuell muss der Nutzer eine Lösung auswählen, die für die Bytecodegenerierung verwendet werden soll. Durch die Auswahl einer Lösung wird jedoch der Funktionsumfang der Klasse eingeschränkt. Besser wäre es alle Lösungen zu beachten und bei Konflikten, diese durch Algorithmen zu lösen. Deshalb soll in dieser Arbeit die vorhandene Bytecodegenerierung angepasst werden, sodass diese alle Lösungen der Typinferenz verwendet. Ein Beispiel für eine Methode mit mehreren Lösungen wird in Listing 3.1 dargestellt. In diesem wird die Addition ausgeführt. Diese ist in Java für die numerischen Typen, aber auch für Strings definiert. Daher kann der Parameter a die folgenden Typen annehmen.

1. Integer
2. Long
3. Float
4. Double
5. String

```
1 public class Beispiel {
2   public method(a){
3     return a+a;
4   }
5 }
```

Listing 3.1: Methoden mit nicht eindeutiger Typisierung

Bei der Verwendung mehrerer Lösungen tauchen jedoch vermutlich einige Probleme auf die es zu beheben gibt. Ein Problem ist beispielsweise das generieren von doppelten Methoden. Wird vom Compiler eine Methode mit dem selben Namen und den selben Parametertypen mehrmals generiert, so wird die JRE diese Klasse nicht verwenden können und die Ausführung mit dem Fehler *java.lang.ClassFormatError: Duplicate method name&signature in class file XYZ* beenden. Diese Probleme treten meist auf, wenn eine Methode Seiteneffekte besitzt. Ein Beispiel dafür ist die Methode *getZaehler* die im Listing 3.2 dargestellt ist. Diese Methode erhöht einen Zählerstand und gibt diesen zurück. Der Zählerstand kann jeden numerischen Typ annehmen, weshalb die Typinferenz alle numerischen Typen als Lösung generieren würde. Generiert man jedoch für jeden numerischen Typ die selbe Methode, ist diese mehrfach vorhanden und kann nicht unterschieden werden.

```
1 public class Beispiel {
2   zaehlvariable;
3
4   public getZaehler(){
5     zaehlvariable++;
6     return zaehlvariable;
7   }
8 }
```

Listing 3.2: Methode mit Seiteneffekt

Ein weiteres Problem stellen die generischen Parameter dar. Da die JRE kein dynamisches Typsystem besitzt, werden generische Parameter beim Compilieren durch die Typlöschung (Type Erasure) gelöscht. So werden Typeparameter durch den Type *Object* ersetzt oder vollständig gelöscht. Nach der Typlöschung sind die zusätzlichen Typinformation nicht mehr vorhanden. Durch die Typlöschung ist im offiziellen Standard von Java das Überladen von Methoden mit generischen Parametern nicht möglich, da die JRE die Methoden nicht mehr unterscheiden kann.

Da dieses Forschungsprojekt sich rund um Typen dreht, und die generischen Parameter eine zentrales Konzept des Typsystem von Java bildem, muss es möglich sein diese zur Laufzeit zu unterscheiden.

```
1 import java.util.Vector;
2
3 public class Beispiel {
4   public void method(Vector<String> vector){
5     ...
6   }
7
8   public void method(Vector<Integer> vector){
9     ...
10  }
11 }
```

Listing 3.3: Überladung von Methoden mit generischen Typen

4 Konzept

In der Problemstellung wird schon deutlich, dass es nicht damit getan ist, blind drauf los zu programmieren. Daher wird in diesem Kapitel ein Konzept entwickelt und erläutert, um die Probleme des Überladens zu lösen.

4.1 Verwendung aller Typisierungen

Das erste Ziel dieser Arbeit ist es, möglichst alle Lösungen der Typinferenz hinzuzuziehen um eine möglichst allgemeine Klasse zu generieren. Aktuell muss der Nutzer aus allen gültigen Lösungen, die von ihm gewünschte Lösung auswählen. In Zukunft soll der Nutzer diese Auswahl nicht mehr treffen müssen. Die Auswahl soll automatisch durch den Compiler getroffen werden.

Die Umsetzung soll iterativ geschehen, das heißt, dass das die Lösung Schritt für Schritt erarbeitet wird. Um die allgemeinste Klasse zu erhalten sollen im ersten Schritt alle Typisierungen als Methoden umgesetzt werden, sodass jede Methode so viele Überladungen, wie Typisierungen besitzt. Diese Umsetzung hat einige Schwächen, die aber Schritt für Schritt verhindert werden sollen.

Das erste Problem, der Verwendung von allen Lösungen, wird schnell deutlich.. Die JVM unterscheidet die Methoden anhand den Methodennamen und dem Deskriptor. Unterscheiden diese sich nicht, so kann die JVM keine eindeutige Entscheidung treffen und beendet damit das Programm mit einem Fehler. Wie in dem Kapitel 3 erläutert, kommt es aber sehr schnell zu einer Methode mit gleichen Typisierungen, da sich eine Typisierung nicht auf eine Methode, sondern auf die ganze Klasse bezieht.

Duplizierte Methoden können nur auf zwei Arten verhindert werden. Entweder man verarbeitet die Typisierungen und entwickelt einen Algorithmus, der doppelte Methoden verhindert, oder man erstellt die Methoden und überprüft bevor man diese generiert, ob schon eine Methode mit dem selben Namen und der selben Typisierung generiert wurde.

Das vorherige Filter von doppelte Methoden hätte den Vorteil, dass man die Auswahl der Typisierungen intelligenter treffen könnte. Bei der Überprüfung der schon generierten Methoden, wird immer die Methode verwendet, die als erstes generiert wurde. Auch wenn die zweite vielleicht besser wäre. Doch das vorherige Filtern von doppelten Methoden ist kaum umsetzbar, da die Typisierungen für sich nicht im Kontext zum AST stehen. Dadurch ist es weder möglich zu entscheiden, welche Typisierung zu welcher Methode gehört, noch ist es möglich den Namen der Methoden zu erhalten. Daher müsste das vorherige Filtern durch die Verwendung von AST und aller Typisierungen durchgeführt werden, was sich nicht als praktikabel erweist.

Daher soll die Verhinderung von doppelten Methoden mit der zweiten Strategie umgesetzt werden. Dafür soll gespeichert werden, welche Methoden mit welchem Namen und welchen Parametertypen

schon generiert wurden. Diese können verwendet werden, um vor der Generierung einer Methode zu überprüfen, ob diese generiert werden darf. Dadurch kann sichergestellt werden, dass keine doppelten Methoden generiert werden.

4.2 Generische Datentypen

Ein weiteres Problem stellen die generischen Datentypen dar. Im offiziellen Standard werden die generischen Datentypen durch die Typlöschung vereinfacht, da die JVM keine Typparameter kennt. Durch die Typlöschung werden jedoch alle Typparameter aus der Deskription entfernt, sodass sich die Typen nicht mehr unterscheiden lassen.

Im aktuellen Standard von Java wird eine Signatur aufgebaut, wie sie in Listing 4.1 dargestellt ist. Das große L am Anfang zeigt auf, dass der Typ eine Klasse ist und kein primitiver Datentyp oder ein Array. Nach dem L folgt der qualifizierte Klassennamen, wobei alle Punkte in ein Slash umgewandelt werden. Beendet wird jede Signatur durch ein Semikolon.

```
1 Ljava/lang/String;
```

Listing 4.1: Standard Java Signatur eines Strings

Gibt es einen oder mehrere Parameter für die Klasse, so werden diese auch im aktuellen Standard in die Signatur codiert. Ein Beispiel dafür ist Listing 4.2, in dem eine Signatur der Containerklasse Vektor mit dem generischen Parameter String dargestellt ist. Der Parameter String wird nach dem Klassennamen Vektor in der Signatur codiert. Dafür wird die Signatur der Klasse String zwischen einem Kleiner-Als- und einem Größer-Als-Zeichen nach dem Klassennamen Vektor angefügt.

```
1 Ljava/util/Vector<Ljava/lang/String;>;
```

Listing 4.2: Standard Java Signatur eines Vektors aus Strings

Alle Informationen zwischen dem Kleiner-Als- und dem Größer-Als-Zeichen werden durch die Typlöschung entfernt. Deshalb hat die JRE zur Laufzeit keine Informationen über die Parameter. Sodass eine Methode, die einen einem Parameter mit generischen Parametern besitzt, nicht überladen werden kann. Eine Klasse wie in Listing 3.3 dargestellt, ist nicht möglich, da die Deskriptoren der beiden Methoden sich nicht unterscheiden.

Um dies zu verhindern werden die Parameter nicht mehr als zusätzliche Informationen in die Signatur codiert, sondern in den Klassennamen. Das neue Signaturformat ist ist Listing 4.3 dargestellt. Statt durch Slashes und den Kleiner- und Größer-Als-Zeichen wird die Trennung durch ein bzw. zwei Prozentzeichen durchgeführt. Da das Prozentzeichen als valides Zeichen einer Klasse in der JVM angesehen wird, können so die Klassen mit Parametern als eigenständige Klassen modelliert werden. Diese Sytnax ist angelehnt, an die Syntax die in „Another Extension of the Java Type System“ [Plü15a] vorgestellt wurde.

```
1 Ljava%util%Vector%%java%lang%String%;
```

Listing 4.3: Neue Signatur eines Vektors aus Strings

Durch diese Änderung sind Typen mit Parameter unterscheidbar, da jede Parameterkombination als eigenständige Klasse repräsentiert wird. Da diese Typen im Deskriptor von Methoden verwendet werden, sind auch diese voneinander unterscheidbar. Die JRE kann nun die überladenen Methoden aus dem Beispiel aus dem Listing 4.3 unterscheiden.

Damit die JRE die Typen auch kennt, müssen für alle verwendeten Parameterkombinationen eine Klasse generiert werden, die den neuen Namen trägt, sodass diese mit dem Classloader geladen werden können. Diese Klassen haben jedoch kein Methoden oder Variablen. Damit sie die Funktionen der eigentliche Klasse ausführen können, sollen diese von dieser Klasse erben. Dadurch können diese alle Funktionen der Klasse ausführen.

Um das Überladen von Methoden mit generischen Typen zu ermöglichen sind vier Schritt notwendig. Das Ziel ist es die generischen Parameter von Methodenparameter in dem Methodendeskriptor zu codieren. Dies soll erreicht werden, indem der Deskriptor von Klassen angepasst wird. Dieser soll mithilfe der vorgestellten Syntax gelöst werden. Damit die JRE die Klassen kennen kann, müssen diese als eigenständige Klasse im Bytecode umgesetzt werden. Im letzten Schritt sollen die neu erzeugten Klassen von der ursprünglichen Klasse erben, sodass sie den selben Funktionsumfang bieten.

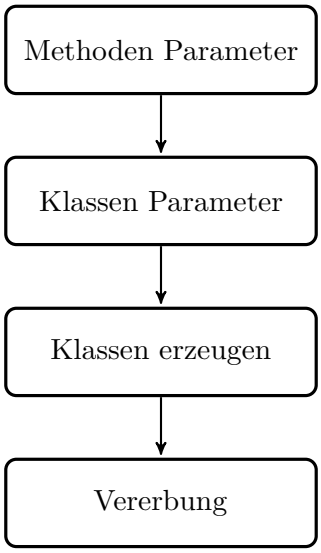

Abbildung 4.1: Konzept für generische Datentypen

Die JVM muss für diese Änderung nicht angepasst werden. Sie erkennt die vollständige Typangabe als eigenständige Klasse. Damit die JVM, diese als Klasse erkennen kann, müssen die neu eingeführten Klassen als eigenständige Klassen im Bytecode umgesetzt werden.

5 Umsetzung

Die Umsetzung der Bytecodegenerierung in dieser Arbeit wurden iterativ durchgeführt. Nachdem die aktuelle Bytecodegenerierung analysiert wurde, wurden die Änderungen in mehreren Iterationen durchgeführt. Die Analyse und die Iterationsschritte werden in diesem Kapitel beschrieben.

5.1 Vorarbeit

Für die Bytecodegenerierung wird im Java-TX Projekt der abstrakte Syntaxbaum und das Ergebnis der Typinferernz verwendet. Diese werden mit der Methode *generateBytecode* der Klasse *MyCompiler* übergeben, die diese auswertet und den Bytecode zurückgibt.

Dafür wird in der *generateBytecode* Methode des Compilers die *generateBytecode* Methode der untersten Elemente des AST, den *SoureFiles* aufgerufen. Diese erzeugen den Bytecode für eine Datei, die mehrere Klassen enthalten kann. Die einzelnen Klassen werden wiederum durch den Aufruf der Knoten im AST erzeugt, die die Klassen repräsentieren. Der Bytecode entsteht nach und nach durch Aufrufe von den *generateBytecode* der einzelnen AST-Knoten.

Die Typen, die angepasst werden müssen, können zum einem Knoten des AST sein, zum anderen können sie im Ergebnis der Typeinferenz enthalten sein, wenn der Typ im AST ein *TypePlaceholder* ist. Die Klasse *TypePlaceholder* repräsentiert einen Type, der nicht vom Programmierer explizit angegeben wurde, sondern der durch die Typeinfernenz berechnet werden soll. Alle Typen leiten von der abstrakten Klasse *Type* ab, die in Listing 5.1 vereinfacht dargestellt ist. Durch den Aufruf der Methode *getBytecodeSignature* erhält, erhält man die Signatur eines Typs als String. Diese Methode muss für einige Typen angepasst werden.

```
1  public abstract class Type extends SyntaxTreeNode{
2    public abstract BcelType getBytecodeType(ClassGenerator cg,
         TypeinferenceResultSet rs);
3
4    /**
5     * Erzeugt einen String, welcher den Typ genauer angibt.
6     * Dieser kann dann in Methoden und Feldersignaturen verwendet werden
7     */
8    public String getBytecodeSignature(ClassGenerator cg, TypeinferenceResultSet
         rs) {
9      return this.getBytecodeType(cg, rs).getSignature();
10   }
11 }
```

Listing 5.1: vereinfachte abstrakte Klasse *Type*

Für die Bytecodegenerierung von überladenen Methoden sind Variablen und Operatoren nicht direkt notwendig, zum Testen der Funktionalität aber sehr sinnvoll. Da diese fehlerhaft und unvollständig aktuell implemenitert sind, wurde die Funktionalität der Variablen und aller Operatoren implementiert, was im folgenden erläutert wird.

5.1.1 Variablen

in Java gibt es vier verschiedene Arten von Variablen. Es gibt statische Klassenvariablen, Objektvariablen, Parameter und lokale Variablen.

Die Parameter einer Methode und lokale Variablen liegen auf dem Stack, wogegen Klassen und Objektvariablen auf dem Heap liegen. Für die Variablen auf dem Stack gibt es die Befehle *ASTORE* und *ALOAD*, mit deren Hilfe man Objekte vom Stack laden und auf dem Stack speichern kann. Für diese Befehle wird jeweils ein Index benötigt, der auf den Platz des Stacks zeigt, von dem gelesen bzw. zu dem geschrieben werden soll. Um die Variablen anhand des Namens zu identifizieren wird deshalb eine Relation benötigt, die dem Namen einer Variable deren Index auf dem Stack zuweist. Dies wurde in der Klasse *DHBWInstructionFactory* umgesetzt, die in Listing 5.2 dargestellt ist. In dieser gibt es eine *HashMap*, welche den Namen als Schlüssel und den Index als Wert besitzt. Wenn eine Variable gesetzt werden soll, so kann über die Methode *createStore*, die den Type der Variable und dessen Namen benötigt, verwendet werden um die Bytecodebefehle zu generieren. Intern wird durch den Variablename mithilfe der *HashMap* auf den Index zugegriffen. Mit dem Index und dem Variabletypen kann man mithilfe der Methode *createStore* der Klasse *InstructionFactory* von BCEL, von der die *DHBWInstructionFactory* geerbt hat, verwenden. Diese erzeugt dann den eigentlichen Bytecode. Das Laden einer Variable erfolgt analog unter Verwendung der *createLoad*-Methode.

```
1  public class DHBWInstructionFactory extends InstructionFactory{
2    private static Map<String, Integer> storeIndexes = new HashMap<>();
3
4    ...
5
6    public LocalVariableInstruction createLoad(org.apache.commons.bcel6.generic.
         Type bytecodeType, String variableName) {
7      return InstructionFactory.createLoad(bytecodeType, getStoreIndex(
           variableName));
8    }
9
10   public LocalVariableInstruction createStore(org.apache.commons.bcel6.generic.
         Type bytecodeType, String variableName) {
11     return InstructionFactory.createStore(bytecodeType, getStoreIndex(
           variableName));
12   }
13
14   public Integer getStoreIndex(String variableName) {
15     if(storeIndexes.get(variableName) == null){
16       Integer index = storeIndexes.size()+1;
17       storeIndexes.put(variableName, index);
18     }
```

```
19
20      return storeIndexes.get(variableName);
21    }
22  }
```

Listing 5.2: Ausschnitt der Klasse DHBWInstructionFactory

Das Laden und Speichern von Variablen wird bei der Bytecodegenerierung von einigen AST-Elementen benötigt. Durch den Einsatz einer zentralen Factoryklasse, ist es möglich die Zustände über Diese konsistent zu halten. Ein Beispiel für das Speichern von Daten auf dem Stack, ist der AST-Knoten *Assign*. Die Generierung dieses Knotens wird in Listing 5.3 dargestellt. Die eigentliche Generierung erfolgt durch das Auswerten der Expression, die rechts des Gleichzeichens steht, durch den Aufruf dessen *genByteCode*-Methode, sowie dem Aufruf der *createStore*-Methode mit dem Variablenamen.

```
1 @Override
2 public InstructionList genByteCode(ClassGenerator cg, TypeinferenceResultSet rs)
      {
3     DHBWInstructionFactory _factory = new DHBWInstructionFactory(cg, cg.
          getConstantPool());
4     InstructionList il = expr2.genByteCode(cg, rs);
5
6     il.append(_factory.createStore(expr2.getType().getBytecodeType(cg, rs),
          expr1.get_Name()));
7
8     return il;
9 }
```

Listing 5.3: Die Methode genByteCode der Klasse Assign

5.1.2 Operatoren

Operatoren sind Symbole der Syntax, welche als Abkürzung für eine Funktion stehen. Es gibt beispielsweise den Plus-Operator, der Zahlen addiert. Diese Operatoren waren so umgesetzt, dass der Bytecode der erzeugt wurde, für jeden Operator, die Gleichheit von zwei 32-Bit Ganzzahlen überprüft hat. Die Bytecodegenerierung der Operatoren wurde deshalb implementiert, sodass beispielsweise Vergleiche möglich sind. Außerdem wurde bei der Implementierung darauf geachtet, dass diese nicht nur mit primitiven Datentypen funktionieren, sondern auch mit den dazugehörigen Wrapper-Klassen. Dies kann man als Vorstufe für das Auto- bzw. Unboxing ansehen.

Die Erzeugung des Bytecodes erfolgt in der Methode *genByteCode* der Klasse *Binary*. In dieser Methode hat man Zugriff auf den linken und rechten Operanden, sowie dem Operator. Um den Quellcode übersichtlich zu gestalten, wird die Bytecodegenerierung nicht direkt in dieser Methode ausgeführt, stattdessen wurde eine neue Methode *genByteCode* in der Klasse *Operator* eingeführt, die für die Bytecodegenerierung des speziellen Operators zuständig ist. Die Methode *genByteCode* der Klasse *Binary* delegiert die Generierung auf diese Klassen weiter.

Die eigentliche Generierung erfolgt beispielsweise in der Klasse *OrOp*, die für den Oder-Operator steht. Mithilfe von BCEL wird der Bytecode erzeugt. Der erforderliche Bytecode ist in Listing 5.4 und die dazugehörige *genByteCode*-Methode in 5.5 dargestellt.

```
1  0: aload_1
2  1: invokevirtual #2          // Method java/lang/Boolean.booleanValue:()Z
3  4: ifne          14
4  7: aload_2
5  8: invokevirtual #2          // Method java/lang/Boolean.booleanValue:()Z
6  11: ifeq          18
7  14: iconst_1
8  15: goto          19
9  18: iconst_0
10 19: invokestatic  #3          // Method java/lang/Boolean.valueOf:(Z)Ljava/lang/
       Boolean;
```

Listing 5.4: Bytecode des OR Operators

Es werden zwei Boolean-Objekte erwartet. Diese werden mit *aload_n* geladen und mit *invokevirtual* in den primitiven Datentype bool umgewandelt. Durch *invokevirtual* wird die *booleanValue* der Objekte aufgerufen. Die eigentliche Auswertung wird durch die Sprungbefehle *ifne* und *ifeq* ausgeführt. *Ifeq* prüft ob der oberste Wert auf dem Stack die Zahl Null ist und *ifne* prüft ob diese nicht Null ist. Da die booleschen Werte im Bytecode als die Intergerwerte 0 und 1 repräsentiert werden, können diese Funktionen verwendet werden.

In der Zeile 4 wird überprüft ob der erste Wert nicht Null ist, als den booleschen Wert true ist. Trifft dies zu so wird zu Zeile 14 gesprungen, die Zahl 1 auf den Stack gelegt und mit *goto* zur Zeile 19 gesprungen. Ist der erste Wert Null, so wird der zweite Wert geladen und es wird überprüft, ob dieser 0, als false ist. Tritt dies wiederum auf, so wird zu Zeile 18 gesprungen und die Zahl 1 auf den Stack geladen. Anderenfalls wird die Zahl 1 auf den Stack geladen und mit *goto* zur Zeile 19 gesprungen. Egal ob eine 1 oder eine 0 auf dem Stack liegt, als letztes wird immer die Zeile 19 ausgeführt. Mit diesem Aufruf von *invokevirtual* wird der boolesche Wert bzw. die Zahl 1 oder 0 mit der statischen Methode *valueOf* der Klasse *Boolean* in ein Objekt umgewandelt.

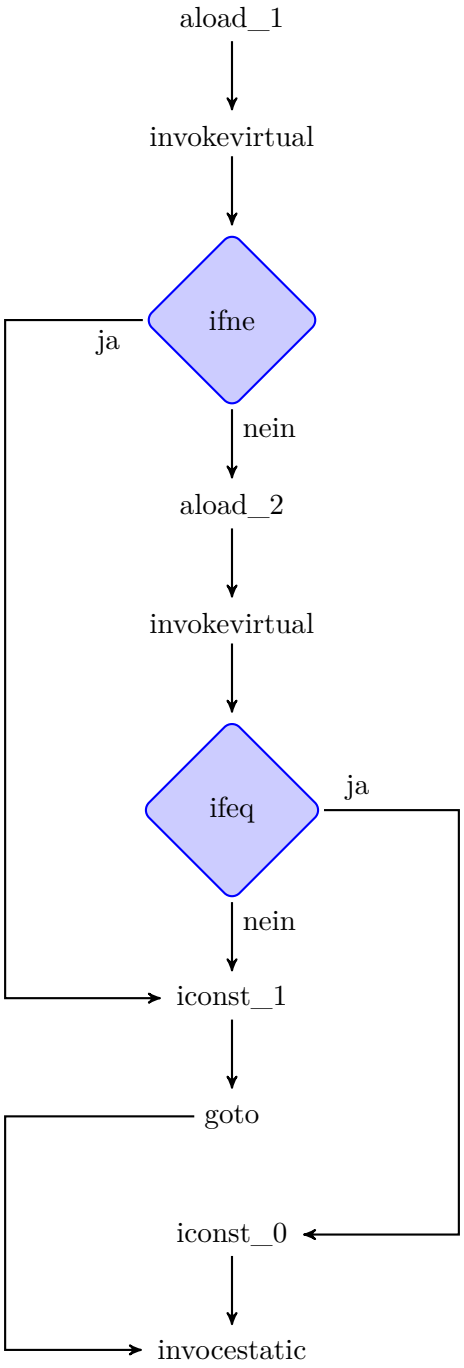

Abbildung 5.1: OR Operator

```
1  public InstructionList genByteCode(ClassGenerator _cg, TypeinferenceResultSet rs
       , Binary operator) {
2    DHBWInstructionFactory _factory = _cg.getInstructionFactory();
3
4    InstructionList il = operator.get_Expr1().genByteCode(_cg, rs);
5    il.append(_factory.createInvoke(
6      "java.lang.Boolean", "booleanValue",
7      org.apache.commons.bcel6.generic.Type.BOOLEAN,
8      new  org.apache.commons.bcel6.generic.Type[] {},
9      Constants.INVOKEVIRTUAL)
10   );
11
12   BranchInstruction firstTest = new IFNE(null);
13   il.append(firstTest);
14
15   il.append(operator.get_Expr2().genByteCode(_cg, rs));
16   il.append(_factory.createInvoke(
17     "java.lang.Boolean", "booleanValue",
18     org.apache.commons.bcel6.generic.Type.BOOLEAN,
19     new  org.apache.commons.bcel6.generic.Type[] {},
20     Constants.INVOKEVIRTUAL)
21   );
22
23   BranchInstruction secondTest = new IFEQ(null);
24   il.append(secondTest);
25
26   firstTest.setTarget(il.append(InstructionConstants.ICONST_1));
27
28   BranchInstruction gotoInstruction = new GOTO(null);
29   il.append(gotoInstruction);
30
31   secondTest.setTarget(il.append(InstructionConstants.ICONST_0));
32
33   gotoInstruction.setTarget(
34     il.append(_factory.createInvoke(
35       "java.lang.Boolean", "valueOf",
36       new ObjectType("java.lang.Boolean"),
37       new  org.apache.commons.bcel6.generic.Type[]{
38         org.apache.commons.bcel6.generic.Type.BOOLEAN},
39       Constants.INVOKESTATIC))
40   );
41
42   return il;
```

Listing 5.5: Bytecodeerzeugung des OR Operators

Die Umsetzung des Bytecodes ist in Listing 5.5 dargestellt. Sie erfolgt analog zum eigentlichen Bytecode.

Mit der Methode *genByteCode* der Klasse *Expression* wird der Bytecode der Operanden erzeugt. Dies kann wie im Beispiel das einfach Laden einer Variable, aber auch das Ergebnis verschachtelter Bedingungen sein. Erwartet wird, dass die Operanden vom Type Boolean sind. Von diesem

wird der primitive Datentyp bool mithilfe der *createInvoke*-Methode abgefragt. Diese hat fünf Parameter. Der erste Parameter ist der Type der Klasse, auf der die Methode aufgerufen werden soll. Der Zweite ist der Methodenname, in diesem Fall *booleanValue*. Im dritte Parameter wird der Rückgabetype der Methode und im Vierten die Typen der Argumente festgelegt. Der letzte Parameter legt fest, ob die Methode auf einem Objekt aufgerufen wird oder statisch.

Die Sprungbefehle *ifne*, *ifeq* und *goto* sind in BCEL als Klassen modelliert. Der Konstruktorparameter ist das Ziel des Sprungs. Dieser kann aber meist nicht direkt gesetzt werden, sondern erst im Nachhinein durch die Methode *setTarget*.

Das Laden der Zahl 1 oder 2 wird durch eine statische Konstante in BCEL gelöst.

Der letzte Aufruf von *createInvoke* erzeugt den Bytecode, der den primitiven Datentyp bool in das Objekt von Type Boolean umwandelt. Dies erfolgt durch die statische Methode *valueOf*, was am letzten Parameter *Constants.INVOKESTATIC* erkennbar ist.

5.2 Jede Typisierung generieren

Im ersten Interationsschritt wurde für jede Lösung der Typeinferenz eine eigene Methode generiert. Dadurch werden Methoden automatisch für alle Lösungen überladen. Auch wenn klar ist, dass dies keine vollständige Lösung der Problemstellung ist, da dadurch doppelte Methoden auftreten können, ist es ein großer Schritt in die richtige Richtung. Um dies umzusetzen, wurde das Generieren von Methoden angepasst, sowie ein Konzept entwickelt, wie die Generierung von Bytecode effektiv getestet werden kann. Diese Änderungen werden in diesem Abschnitt beschrieben.

5.2.1 Änderung der Methodengenerierung

Die Methodengenerierung musste soweit angepasst werden, sodass sie für jede Lösung der Typinferenz eine Methode generiert. Dafür musste die Methode *genByteCode* der Klasse *Method* angepasst werden. Sie erhält statt einem *TypeinferenceResultSet* eine Menge von *TypeinferenceResultSet* als *TypeinferenceResults*, sodass auf alle Typeinferenzlösungen zugegriffen werden kann. Dies wurde umgesetzt, indem der *ClassGenerator* Klasse, die für die Generierung des Bytecodes verwendet wird, als Klassenvariable die Klasse *TypeinferenceResults* hat.

Bei der Generierung des Bytecodes für Methoden, wird über alle Typeinferenzlösungen iteriert und für jede Lösung die Generierung durchgeführt. Dies wird in Listing 5.6 dargestellt. Pro Typeinferenzlösung wird in der Methode *addMethodToClassGenerator* eine Methode als Bytecode generiert.

```
1 public void genByteCode(ClassGenerator cg, Class classObj) {
2   List<TypeinferenceResultSet> typeInterferenceResults = cg.
        getTypeinferenceResults().getTypeReconstructions(this, cg);
3   DHBWInstructionFactory _factory = cg.getInstructionFactory();
4
5   for(TypeinferenceResultSet t: typeInterferenceResults){
```

```
6       addMethodToClassGenerator(cg, _factory, t);
7     }
8   }
```

Listing 5.6: Methode genByteCode der Klasse Method

Bevor der Bytecode durch das BCEL-Framework erzeugt wird, muss die Argumentliste erzeugt werden. Besitz eine Methode mindestens ein Argument, so wird die Argumentliste durch die Methode *generateArgumentList*, die in Listing 5.8 dargestellt ist, erzeugt. Zusätzlich zur Argumentliste wird die `ACCESS_FLAG` mithilfe der Variable `constants` modelliert. Aus diesen Angaben und der Angabe des Rückgabetypes wird ein *MethodenGenerator* erzeugt, der diese Methode erzeugen kann. Die durch *createMethod* erzeugt Methode wird dem *ClassGenerator* übergeben, der die vollständige Klasse generiert.

```
1  private void addMethodToClassGenerator(ClassGenerator cg, DHBWInstructionFactory
       _factory, TypeinferenceResultSet t) {
2    DHBWConstantPoolGen _cp = cg.getConstantPool();
3    InstructionList il = new InstructionList();
4
5    org.apache.commons.bcel6.generic.Type[] argumentTypes =  org.apache.commons.
       bcel6.generic.Type.NO_ARGS;
6    String[] argumentNames = new String[]{};
7    if(this.parameterlist != null && this.parameterlist.size() > 0){
8      generateArgumentList(argumentTypes, argumentNames, cg, _factory, t);
9    }
10
11   short constants = Constants.ACC_PUBLIC;
12   if(this.modifiers != null && this.modifiers.includesModifier(new Static()))
       constants += Constants.ACC_STATIC;
13
14   Type returnType = this.getType();
15
16   MethodGenerator method = new MethodGenerator(constants, returnType.
       getBytecodeType(cg, t), argumentTypes , argumentNames, this.
       get_Method_Name(), getParentClass().name, il, _cp);
17
18   cg.addMethod(method.createMethod(cg, getParameterList(), returnType, get_Block
       (), t));
19 }
```

Listing 5.7: Methode addMethodToClassGenerator der Klasse Method

Die Methode *generateArgumentList* wird verwendet um eine nicht leere Argumentliste zu generieren. Dafür wird über alle Argumente, die im AST festgelegt sind, iteriert. In der Schleife werden die Argumentnamen und -typen bestimmt. Zusätzlich wird in der *DHBWInstructionFactory* mithilfe des Parameternamen ein Platz auf dem Stack für den Parameter reserviert.

```
1  private void generateArgumentList(org.apache.commons.bcel6.generic.Type[]
       argumentTypes, String[] argumentNames, ClassGenerator cg,
       DHBWInstructionFactory _factory, TypeinferenceResultSet t) {
2    argumentTypes = new org.apache.commons.bcel6.generic.Type[this.parameterlist.
       size()];
```

```
3   argumentNames = new String[this.parameterlist.size()];
4   int i = 0;
5   for(FormalParameter parameter : this.parameterlist){
6     argumentTypes[i] = parameter.getType().getBytecodeType(cg, t);
7     argumentNames[i] = parameter.getIdentifier();
8     _factory.getStoreIndex(parameter.getIdentifier());
9     i++;
10  }
11 }
```

Listing 5.8: Methode generateArgumentList der Klasse Method

Um die die Änderungen des Projektes sinnvoll zu testen, wurden JUnit Tests entwickelt. Die Entwicklung dieser wird im nächsten Abschnitt beschrieben.

5.2.2 JUnit Tests

Um die Funktionalität der Bytecodegenerierung zu testen, wurden vor dieser Arbeit UnitTests mithilfe von JUnit erstellt. Dieses Tests haben jedoch nur überprüft, ob der Compiler einen Fehler liefert oder nicht. Es war weder möglich Tests zu entwickeln, die nur die Bytecodegenerierung testen, noch wurde der eigentliche Bytecode überprüft. Diese Tests wurden überarbeitet, sodass die Funktionalität auch bei zukünftigen Änderungen sichergestellt werden kann. Dafür wurden zwei abstrakte Tests für die Überprüfung des Bytecodes entwickelt.

Die erste Art von Tests sind die *ASTBytecodeTests*. Mit diesen wird die Bytecodegenerierung mit einem AST und einem *TypeinferencResultSet* aufgerufen. Dadurch sind die Tests unabhängig vom Parser, der Unifikation und der Typeinferenz. Dies macht die Tests viel schneller als Tests, die alle Phasen des Compilers durchlaufen müssen. Ein Nachteil dieser Tests ist, dass sie relativ schwierig zu entwickeln sind. Mithilfe der *ASTFactory* kann man schon relativ einfach ein AST aufbauen. Ein *TypeinferencResultSet* ist dagegen relativ schwierig manuell zu erzeugen.

```
1  public abstract class ASTBytecodeTest {
2    protected Class getClassToTest(){...}
3
4    public SourceFile getSourceFile() {...}
5
6    public TypeinferenceResults getResults() {...}
7
8    public String getRootDirectory() {...}
9
10   public String getTestName() {...}
11 }
```

Listing 5.9: Schnittstelle des abstrakten ASTBytecodeTest

Die wichtigen Methoden eines *ASTBytecodeTest* ist in Listing 5.9 dargestellt. Die öffentlichen Methoden werden benötigt um die Test zu konfigurieren. Mit der Methode *getSourceFile* wird eine *SourceFile* zurückgegeben, welche die zu testende Klasse beinhält. Der AST in der *SourceFile* kann mithilfe der *ASTFactory* generiert werden. Mit *getRootDirectory* legt man den Ordner

fest, in dem die Testklasse mit dem Namen, die die Methode *getTestName* zurückgibt, generiert wird.

Der schwierige Teil dieser Art von Test ist die *getResults*-Methode. Mit dieser werden *TypeinferenceResults* zurückgegeben, die in Kombination mit dem AST die zu testende Klasse ergeben. Für das Generieren von *TypeinferenceResults* gibt es noch keine einfache Methode. Es müssen manuell alle Lösungen zusammengebaut werden.

```
1  public class ExtendsVectorTest extends ASTBytecodeTest{
2    public SourceFile getSourceFile(){
3      /*
4      import java.util.Vector;
5
6      class ExtendsVector extends Vector<Object>{
7
8      }
9       */
10     SourceFile sourceFile = new SourceFile();
11
12     de.dhbwstuttgart.syntaxtree.Class classToTest = ASTFactory.createClass(
           getTestName(), new RefType("java.util.Vector", sourceFile, 0), null,
           null, sourceFile);
13     sourceFile.addElement(classToTest);
14
15     return sourceFile;
16   }
17
18   @Test
19   public void testSupertype(){
20     try{
21       ClassLoader classLoader = getClassLoader();
22
23       Class cls =  classLoader.loadClass(getTestName());
24
25       assertEquals("java.util.Vector", cls.getSuperclass().getName());
26     }catch(Exception e){
27       e.printStackTrace();
28       fail();
29     }
30   }
31
32   @Override
33   public String getRootDirectory() {
34     return super.getRootDirectory()+"types/";
35   }
36
37
38
39   @Override
40   public String getTestName() {
41     return "ExtendsVector";
42   }
```

```
43
44 }
```

Listing 5.10: Beispielimplementierung eines ASTBytecodeTest anhand des ExtendsVectorTest

Ein Beispiel für ein *ASTBytecodeTest* ist der *ExtendsVectorTest* der in Listing 5.10 dargestellt ist. In der Methode *getSourceFile* wird eine neue *SourceFile* erstellt. Dieser wird mithilfe der *ASTFactory* eine neue Klasse hinzugefügt, die von *java.util.Vector* ableitet und sonst keine Felder oder Methoden besitzt. In der Methode *testSupertype* wird der Supertyp der zu testenden Klasse mithilfe des *ClassLoaders*, der von *getClassLoader* zurückgegeben wird, und Reflektion getestet.

Die zweite Art von Tests sind die *SourceFileBytecodeTests*. Diese durchlaufen den vollständigen Compilerprozess. Dadurch sind sie viel einfacher zu entwickeln, da man die Klassen als Java-Sourcecode ohne Typeangaben programmieren kann.

```
1 public abstract class SourceFileBytecodeTest extends TestCase{
2   public String rootDirectory = System.getProperty("user.dir")+"/test/bytecode/"
        ;
3   public String testFile;
4   protected String testName;
5
6   protected abstract void init();
7
8   protected ClassLoader getClassLoader() throws Exception{}
9 }
```

Die Klasse *SourceFileBytecodeTests* besitzt vier Klassenvariablen, die Einstellungen für den Test vornehmen. Diese müssen in der *init*-Methode festgelegt werden. Mit der Methode *getClassLoader* wird ein *ClassLoader* erzeugt, der Klassen aus dem *outputDirectory* lädt. Dieser kann dazu verwendet werden die Klasse, die man testen möchte zu laden.

```
1  public class VariableTest extends SourceFileBytecodeTest{
2    @Override
3    protected void init() {
4      testName = "Variable";
5      rootDirectory = System.getProperty("user.dir")+"/test/bytecode/";
6    }
7
8    @Test
9    public void testConstruct() throws Exception{
10       ClassLoader classLoader = getClassLoader();
11
12       Class cls =  classLoader.loadClass(testName);
13
14     Object obj = cls.newInstance();
15     assertTrue(true);
16   }
17 }
```

Listing 5.11: Beispielimplementierung eines SourceFileBytecodeTest anhand des VariableTest

Ein Beispiel für einen *SourceFileBytecodeTest* ist der *VariableTest*. Dieser ist in Listing 5.11 und die dazugehörige Jav-Datei in 5.12 dargestellt.

```
class Variable{
  public Integer method(Integer parameter){
    Integer lokaleVariable;
    lokaleVariable = 2;

    return parameter+lokaleVariable;
  }
}
```

Listing 5.12: Testdatei für den VariableTest

Die Jav-Datei enthält in diesem Fall validen Javacode. Es könnten aber auch einige Typen weggelassen werden. Da die Bytecodegenerierung getestet werden soll, ist es aber sinnvoll möglichst viele Typen anzugeben, um unabhängig von der Typeinferenz zu testen.

Im *SourceFileBytecodeTest* werden in der *init*-Methode der Testname und der Ordner, in den die Classfiles generiert werden solle, konfiguriert. Der eigentliche Test wird in der Methode *testConstruct* ausgeführt. In dieser wird die zu testende Klasse geladen und ein neues Objekte erzeugt.

Die zwei Arten von Tests unterscheiden sich dahingegen, welche Teile des Projektes beim Testen ausgeführt und damit getestet werden. Im Folgenden wird beschrieben, wie speziell der Bytecode getestet werden kann. Das Testen von Bytecode erfolgte bis jetzt in diesem Projekt manuell. Es werden JUnit-Tests entwickelt, welche aus einer Jav-Datei mithilfe des vollständigen Projektes eine Class-Datei generiert. Sobald dieser Vorgang ohne Fehler beendet ist, ist die erste Hürde überwunden, doch man kann sich noch nicht sicher sein, dass der Bytecode korrekt funktioniert. Zum einen wird nicht getestet, ob die JRE den Bytecode lesen kann, zum anderen wird die Funktionalität der Bytecodes nicht getestet. Diese Schritte müssen manuell ausgeführt werden, indem man den Bytecode mit dem Bytecode vergleicht, den die JVM erzeugt. Dafür kann beispielsweise der Befehl *javap* verwendet werden, um den Bytecode zu disassablieren. Dieser manuelle Schritt soll in Zukunft automatisiert ausgeführt werden, indem die Funktionalität mithilfe von Reflection getestet wird.

Die erste Art von Fehlern, das erzeugen von ungültigem Bytecode, kann man durch das erzeugen einer neuen Instanz der zu testenden Klasse überprüfen. Das Erzeugen einer neuen Instanz kann mithilfe des *ClassLoaders* durchgeführt werden, der von den beiden neuen Arten von Bytecodetests zur Verfügung gestellt wird. Dies ist in Listing 5.13 dargestellt.

```
public void testConstruct() throws Exception{
  ClassLoader classLoader = getClassLoader();

  Class cls =  classLoader.loadClass(testName);

  Object obj = cls.newInstance();
  assertTrue(true);
}
```

Listing 5.13: Testmethode testConstruct

Mit der Methode *getClassLoader* erhält man eine Instanz eines *ClassLoaders*, welcher Klassen aus dem Ordner laden kann, den man im Test definiert hat. Mit diesem kann man eine Klasse mit *loadClass* und dem Klassennamen laden. Ein neues Objekt dieser Klasse kann mit der Methode *newInstance* erzeugt werden. Enthält der Bytecode falsche Instruktionen, so wird beim Laden der Klasse oder beim Erzeugen einer neuen Instanz der Test mit einer Exception beendet, welche den Fehler näher erläutert. Anderenfalls wird der Test ohne Fehler beendet.

Mit Reflektion kann jedoch nicht nur ungültiger Bytecode erkannt werden, sondern auch die Funktionalität von diesem getestet werden. Dargestellt wird dies beispielhaft an dem Plus-Operator. Getestet wird dies durch die Klasse *AddOperator*, welche eine Methode besitzt, die zwei Integers addiert und zurückgibt.

```
1 class AddOperator{
2   Integer method(Integer x, Integer y){
3     return x + y;
4   }
5 }
```

Listing 5.14: Klasse AddOperator zum Testen von Rückgabewerten

Zu beginn des Tests wird, wie im vorherigen Test, eine neue Instanz der zu testenden Klasse mithilfe von *newInstance* erzeugt. Die Funktionalität der Klasse *method* wird jedoch zusätzlich durch den Aufruf und der Auswertung des Rückgabewertes überprüft. Aufgerufen werden kann die Methode mit der *invoke*-Methode eines Methoden-Objektes, welche das davor erzeugte Objekt der Klasse und die Parameter der Methode übergeben bekommt. Der Rückgabewert der *invoke*-Methode, ist der Rückgabewert der aufgerufenen Methode. Das Methoden-Objekt erhält man durch den Aufruf der Methode *getDeclaredMethod* auf dem *Class*-Objekt, welche den Methodennamen und einen Array von *Class*-Objekten benötigt, die die Typen der Methodenparameter repräsentieren. Der eigentliche Test ist der Vergleich des Rückgabewertes der *invoke*-Methode mit dem erwarteten Wert drei.

```
1  @Test
2  public void testTwoIntegers() throws Exception{
3    ClassLoader classLoader = getClassLoader();
4
5    Class cls =  classLoader.loadClass(testName);
6
7    Object obj = cls.newInstance();
8
9    Integer x = new Integer(1);
10   Integer y = new Integer(2);
11
12   Class[] params = new Class[]{
13     x.getClass(),
14     y.getClass(),
15   };
```

```
16
17    Method method = cls.getDeclaredMethod("method", params);
18    Integer returnValue = (Integer) method.invoke(obj, x, y);
19    assertEquals(new Integer(3), returnValue);
20  }
```

Listing 5.15: Test von Rückgabewerten mithilfe von Reflection

5.3 Doppelte Methoden

Nachdem für jede Lösung der Typeinferenz eine eigene Methode im Bytecode generiert wurde, trat ein Fehler auf, dass manche Methoden doppelt im Bytecode vorhanden waren, sodass die JRE manche Programme nicht ausgeführt hat.

Methoden werden von der JRE nur anhand des Namen und des Deskriptors unterschieden. Der Deskriptor enthält dabei die Typen der Methodenparameter. Ist der Name zweier Methoden gleich und die Typen der Methodenparameter unterscheiden sich nicht, kann die Klasse nicht verwendet werden und die JRE verweigert den Aufruf der Klasse mit dem Fehler *„java.lang.ClassFormatError: Duplicate method name&signature in class file“*.

Der Fehler kann beim offiziellen Java durch den Parser abgefangen werden. Möchte man eine Klasse mit zwei Methoden mit dem gleichen Namen und Descriptor parsen, so wird der Parsevorgang mit dem Fehler *„error: method Methodenname() is already defined in Klassenname.“* beendet. Beim erweiterten Java dieses Projektes ist dies nicht mehr möglich, da die Typen und damit der Deskriptor beim Parsevorgang noch nicht eindeutig sind. Typen die nicht angegeben wurden, werden erst mit der Typeinferenz berechnet und während der Bytecodegenerierung verwendet.

Ein Beispiel bei dem mehrere Lösungen in der Typeinferenz auftreten, für die aber nur eine Methode generiert werden soll, ist eine Methode deren Methodenparameter angegeben sind die aber intern lokale Variablen verwendet die durch die Typeinferenz berechnet werden und nicht eindeutig sind. Eine Umsetzung dieses Beispiels ist in Listing 5.16 dargestellt.

```
1   public void successor(Integer x) {
2     y;
3     y = x + 1;
4     System.out.println(y);
5   }
```

Listing 5.16: Methode zur Berechnung und Ausgabe des Nachfolgers

Um das Problem von doppelten Methoden zu lösen, werden bei der Generierung einer Klasse alle schon generierten Methoden gespeichert. Bevor eine zusätzliche Methode generiert wird, wird diese mit den schon vorhandenen Methoden abgeglichen. Der Vergleich der Methoden erfolgt dabei anhand des Namens und der Typen der Methodenparametern. Stimmen diese vollständig überein, so wird die Methode nicht noch einmal generiert, sondern ignoriert.

Diese Funktionalität wurde in der Klasse *ClassGenenerator*, welche eine Ableitung der *ClassGen* Klasse, des BCEL-Frameworks ist, umgesetzt. In dieser Ableitung werden spezifische Änderungen

durchgeführt, die das Java-TX Projekt betreffen. Eines de Änderungen ist, das verhindern der Generierung von doppelten Methoden. Dafür wurde die *addMethod*-Methode überschrieben, die der zu generierenden Klasse eine zusätzliche Methode hinzufügt. Diese überprüft, ob die Methode schon vorhanden ist, indem sie alle schon generierten Methoden in einer Liste hält. Enthält die Liste die zur generierende Methode schon, so wird der Aufruf der *addMethod*-Methode nicht an die darunterliegende Superklasse weitergereicht, sondern unterbrochen.

```
1  public class ClassGenerator extends ClassGen{
2    private List<String> methodsNamesAndTypes = new LinkedList<>();
3
4    ...
5
6    @Override
7    public void addMethod(Method m) {
8      String methodNameAndTypes = m.getName()+Arrays.toString(m.getArgumentTypes()
           );
9
10     if(methodsNamesAndTypes.contains(methodNameAndTypes)){
11       return;
12     }
13
14     methodsNamesAndTypes.add(methodNameAndTypes);
15     super.addMethod(m);
16   }
17 }
```

Listing 5.17: Ausschnitt des ClassGenerators

5.4 Fehlende generische Methoden

Damit die JRE Klassen mit generischen Parametern unterscheiden kann, werden diese in die Signatur des Types codiert, wie es im Abschnitt 4.2 konzipiert wurde, sodass sie nicht von der Typelöschung betroffen werden.

Für die Umsetzung sind einige Änderungen notwendig. Die Änderung der Syntax wird in der Klasse *RefType* umgesetzt, die einen Referenztypen repräsentiert. Für die Änderungen dieser Klasse sind noch zusätzlich Änderungen des *ClassGenerators* notwendig, sowie das Erstellen einer *ASTFactory*. Diese werden in den folgenden Abschnitten erläutert.

5.4.1 Anpassung der Referenztypen

Eine Anpassung der Referenztypen verhindert das Auslöschen von Typinformationen durch Type Erasure. Dafür wird vom offiziellen Format der Signatur abgewichen. Die Typinformationen von generischen Parametern wird in die Signatur kodiert.

Dafür wird das im Kapitel 4 entwickelte Format verwendet. Dieses wird in der Methode *getCombinedType* umgesetzt, die von der *getBytecodeSignature* verwendet wird.

```
public String getCombinedType(ClassGenerator cg, TypeinferenceResultSet rs){
  StringBuilder sb = new StringBuilder();

  if(parameter != null && parameter.size() > 0){
    sb.append(getName().toString().replace(".", "%"));
    sb.append("%%");
    for(Type type: parameter){
      if(type instanceof RefType){
        sb.append(((RefType) type).getCombinedType(cg, rs).replace(".", "%"));
      }else{
        sb.append(type.getBytecodeType(cg, rs).toString().replace(".", "%"));
      }
      sb.append("%");
    }
  }else{
    sb.append(this.getName().toString());
  }
  return sb.toString();
}
```

Listing 5.18: Umsetzung des neuen Signaturformates

Die Methode erzeugt mithilfe eines StringBuilders die Signatur. Enthält der Typ generische Parameter, so wird der qualifizierte Name gefolgt von zwei Prozentzeichen und den zusätzlichen Typinformationen als Signatur verwendet. Die zusätzlichen Typinformationen werden mit einem Prozentzeichen getrennt. Falls deren Typ wiederum eine Referenztype ist, wird dessen *getCombinedType* Methode verwendet, sodass auch mehrfach verschachtelte Typen möglich sind. Alle Punkte in den qualifizierten Klassennamen werden zusätzlich mit der *replace* Methode in Prozentzeichen umgewandelt, sodass die Signatur keine Punkte mehr enthält.

Durch die Anpassung der Signatur würde man von der JRE einen Fehler erhalten, dass die Klasse nicht vorhanden ist. Dieses Problem wird dadurch gelöst, dass für jeden neu erzeugte Signatur eine eigene Klasse generiert wird. Diese Überprüfung erfolgt in der Methode *getBytecodeSignature*. Falls die Klasse generische Parameter enthält bzw. der kombinierte Name nicht dem eigentlichen Klassennamen entspricht wird eine neue Klasse erzeugt. Dafür werden die Klassen *ClassGenerator* und die *ASTFactory* verwendet.

Mithilfe der *ASTFactory* wird ein AST aufgebaut, der dem *ClassGenerator* übergeben wird, sodass dieser die Klasse verwaltet und später generiert. Die zu erzeugende Klasse besitzt den kombinierten Type und leitet von der vorhandenen Klasse ab.

```
public String getBytecodeSignature(ClassGenerator cg, TypeinferenceResultSet rs)
    {
  String combinedType = getCombinedType(cg, rs);
  if(!combinedType.equals(getName().toString())){
    Class generatedClass = ASTFactory.createClass(getCombinedType(cg, rs),
        getGenericClassType(), null, null, new SourceFile());

    cg.addExtraClass(generatedClass.genByteCode(new TypeinferenceResults()).
        getByteCode());
```

```
7   }
8   String ret = new org.apache.commons.bcel6.generic.ObjectType(combinedType).
        getSignature();
9   return ret;
10 }
```

Listing 5.19: getBytecodeSignature des Referenztypes

5.4.2 Änderung des ClassGenerator

Der *ClassGenerator* wurde angepasst, da es möglich sein muss, dass während der Erzeugung einer Klasse eine weitere Klasse hinzugefügt wird, die zusätzlich erzeugt wird. Dies wird benötigt um die %-Klassen zu erzeugen.

Für die Verwaltung der zusätzlichen Klassen wurde im *ClassGenerator* eine Map verwendet, die den Namen der Klassen auf deren *ClassGenerator*-Objekt abbildet. Neue Klassen können mit *addExtraClass* hinzugefügt werden und alle zusätzlichen Klasse können mit *getExtraClasses* abgerufen werden.

Das Hinzufügen einer zusätzlichen Klasse wird beim abfragen der Signatur von *RefTyps* benötigt. Beim Abfragen der Signatur wird eine zusätzliche Klasse benötigt, wenn sie eine %-Klasse ist. Diese wird mithilfe der *ASTFactory* erzeugt und dem *ClassGenerator* übergeben.

Das Abfragen aller zusätzlicher Klasse wird beim Erzeugen der Class-Dateien benötigt. Vor dieser Änderung wurde nur die Class-Datei der eigentlichen Klasse generiert. Mit dieser Änderung muss für jede Klasse die von *getExtraClasses* zurückgegeben werden eine zusätzliche Class-Datei angelegt werden.

5.4.3 Erstellen der ASTFactory

Ein abstrakter Syntaxbaum wird durch den yacc-Parser erstellt. Doch es ist auch nötig außerhalb des Parsers den AST zu erweitern oder einen neuen AST aufzubauen, da dies für automatisierte Test und der Generierung von der Typeklassen benötigt wird. Um dies zu ermöglichen wurde eine *ASTFactory* entwickelt.

Die *ASTFactory* ist eine Klasse mit vielen statischen Methoden, die den Entwickler beim erstellen eines AST zu unterstützen und Fehler zu minimieren. Fehler können dabei minimiert werden, da einige Methodenaufrufe beim Erstellen eines AST automatisch ausgeführt werden, wenn man die Factory verwendet. Ein Beispiel für solch eine Methode ist die *parserPostProcessing*-Methode, welche normalerweise nach dem Parsen aufgerufen werden muss, um Aktionen durchzuführen, die vorher nicht ausführbar sind. Beim manuellen Aufbau eines AST wird dies schnell vergessen, wogegen bei dem Aufbau durch die *ASTFActory* dies automatisch ausgeführt wird.

Zusätzlich zur Fehlervermeidung hat die *ASTFactory* den Vorteil, dass die Erstellung eines AST an einer zentralen Stelle geregelt wird, wodurch Änderungen oder Fehlerbehebungen am AST leichter möglich und weniger aufwendig sind.

Die Methoden die die *ASTFactory* aktuell zur Verfügung stellt sind in Listing 5.20 dargestellt. Es ist sinnvoll, diese Klasse in Zukunft zu erweitern und möglichst viel der Erstellung des AST aus dem übrigen Projekt in diese Klasse zu verlagern, um die Vorteile dieser Klasse zu nutzen.

```
1  public class ASTFactory {
2    public static Method createMethod(String name, ParameterList paralist, Block
         block, Class parent) {}
3
4    public static Method createEmptyMethod(String withSignature, Class parent) {}
5
6    public static Constructor createConstructor(Class superClass, ParameterList
         paralist, Block block){}
7
8    public static Class createClass(String className, Type type, Modifiers
         modifiers, Menge supertypeGenPara, SourceFile parent) {}
9
10   public static Class createObjectClass() {}
11 }
```

Listing 5.20: Schnittstelle der ASTFactory

5.5 Einschränkungen

Bei der Konzeption, der Umsetzung und bei darauffolgenden Tests sind einige Fälle aufgetaucht, die nicht durch die Konzepte abgedeckt werden können, die entwickelt wurden. Die Einschränkungen, die entdeckt wurden, sollen in den folgenden Abschnitten näher erläutert werden.

5.5.1 Wildcards

Wie im Abschnitt 2.2.1 erläutert gibt es in Java verschiedene Arten von Typen. Unter anderen auch die in dieser Arbeit behandelten generischen Datentypen. Diese können jedoch nicht nur mit festgelegten Typparameter verwendet werden, sondern auch mit Wildcards. Diese können die Typparameter in der Vererbungshierarchie einschränken.

Es gibt Unbounded, Upper Bounded und Lower Bounded Wildcards. Unbounded Wildcards werden in der Signatur im offiziellen Standard mit einem „*“ gekennzeichnet. Upper und Lower Bounded Wildcards werden durch eine „+“ bzw. ein „-“ vor dem jeweiligen Typ gekennzeichnet. Im Deskriptor werden alle drei durch die Type Erasure gelöscht.

Genauso wie beim offiziellen Java werden die Typparameter nicht in der Signatur codiert. Es wird nicht die Syntax der %-Klassen verwendet, da sie an dieser Stelle keine Sinn macht. Es ist nicht möglich die in den %-Klassen codierten Werte in der JVM zuzugreifen. Daher ist es beispielsweise, bei der Verwendung eines Upper Bounded Wildcard, nicht möglich die Klasse zu ermitteln, von der abgeleitet werden muss. Außerdem ist die Überprüfung der Vererbung während der Laufzeit nicht möglich.

Die Verwendung der Wildcards muss vollständig vor dem Compilieren überprüft werden. Zur Laufzeit sind keine Überprüfungen mehr möglich. Zusätzlich zu den Problemen der Wildcards, stößt man auf ein Problem, bei der Vererbung von %-Klassen. Dieses wird im folgende Abschnitt erläutert.

5.5.2 Vererbung von %-Klasse

Bei der Vererbung von %-Klassen tritt das Problem der Mehrfachvererbung auf, die Java nicht kennt. Das Problem tritt auf, wenn man eine Vererbungshierarchie mit generischen Parametern hat und man nicht nur die die angegebene Klasse, sondern auch eine Ableitung von dieser verwenden möchte.

Das Problem wird anhand eines Beispieles erläutert. Dieses ist in Listing 5.5.2 dargestellt. Die Methode *first* nimmt eine Liste entgegen und gibt das erste Element der Liste zurück. Diese Methode kann nicht verwendet werden, da der Parameter vom Typ *List* sein muss und dies ein Interface und dadurch nicht initialisierbar ist. In Java kann der Type des Parameters auch eine Ableitung von *List* sein, dies funktioniert jedoch nicht in diesem Projekt. Das Problem wird deutlich, wenn man den Bytecode der Methode betrachtet. Dieser ist in Listing 5.5.2 dargestellt.

```
1 import java.util.List;
2
3 class Mehrfachvererbung{
4         Number first(List<Number> list){
5           return list.get(0);
6         }
7 }
```

In dem Bytecode erkennt man, dass der Typ des Parameters *Ljava%util%List%%java%lang%Number%;* ist. Würde man die Methode mit einem Vektor aufrufen, so wäre dessen Typ *Ljava%util%Vecotr%%java%lang%Number%;*. Diese sind nicht gleich und stehen in keiner Vererbungshierarchie, obwohl *Vector* eine Ableitung von *List* ist. Die Vererbungshierarchie dieses Beispiels wird in der Abbildung 5.5.2 dargestellt.

```
1 public java.lang.Number first(java%util%List%%java%lang%Number%);
2   descriptor: (Ljava%util%List%%java%lang%Number%;)Ljava/lang/Number;
3   flags: ACC_PUBLIC
4   Signature: #14                          // (Ljava%util%List%%java%lang%Number
        %;)Ljava/lang/Number;
5   Code:
6     stack=1, locals=2, args_size=2
7       0: aload_1
8       1: invokevirtual #2                  // Method java/util/List.get:()Ljava/
            lang/Object;
9       4: areturn
```

In dieser Darstellung wird deutlich, dass die beiden Klassen in keiner Vererbungshierarchie stehen. Dies tritt auf, da *List%Number%* von *List* ableitet und *Vector%Number%* von *Vector*, aber die Klasse *Vector* natürlich nicht von *List%Number%*, sondern von *List* ableitet. Dies stellt die größte

Einschränkung dar, da dadurch dieses einfache Beispiel, aber auch sonst viele Beispiele nicht funktionieren würde.

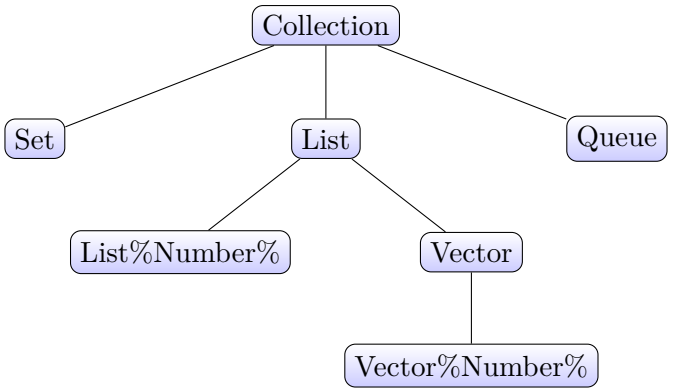

Abbildung 5.2: Beispiel Mehrfachvererbung von Collections

6 Beispiel

In diesem Kapitel wird ein Beispiel vorgestellt und dieses durch den vollständigen Compilevorgang erläutert. Das Beispiel ist eine Klasse welche zwei Methoden besitzt. Beide Methoden besitzen den selben Namen und unterscheiden sich nur durch den Typ des einzigen Parameters.

```
1 class Beispiel {
2   method(x) { return x + x; }
3
4   method(Boolean x) {return x; }
5 }
```

Listing 6.1: Beispielklasse

Der Typ des Parameters der ersten Methode ist nicht festgelegt. Dieser soll durch Typinferenz errechnet werden. Auch der Rückgabetype soll errechnet werden. Der Parametertyp der zweiten Methode ist als Boolean festgelegt. Der Rückgabetyp muss errechnet werden. Die Typeinferenz muss also den Parametertyp und den Rückgabetyp der ersten Methode, sowie den Rückgabetyp der zweiten Methode ermitteln. In der ersten Implementierung wird der Plusoperator auf den Parameter ausgeführt. Der Plusoperator kann in Java eine Addition, sowie eine Konkatenation sein. Die Addition kann bei den numerischen Typen Integer, Long, Float, Double ausgeführt werden. Die Konkatenation bei der Klasse String. Die möglichen Typen sind in diesem Projekt in der Klasse *AddOp* in der Methode *getOperatorTypes* konfiguriert.

Der erste Schritt des Compilevorgangs ist das Lexen und Parsen. Der Parser übersetzt die Jav-Datei in einen AST. Der Wurzelknoten des AST ist eine *SourceFile*, die wiederum eine ein *Class*-Element enthält. In diesem Element sind die beiden Methoden enthalten. Der AST der Beispielmethode mit der Addition ist in der Abbildung 6 dargestellt.

Dieser Teilbaum ist für die Generierung des Bytecodes besonders interessant, da in diesem einige Typen berechnet werden können. Im AST sind die Platzhalter der Typen sichtbar. Diese kann man beim Rückgabetyp, sowie beim Type des Parameters erkennen. Für diese Platzhalter werden mithilfe der Typinferenz gültige Lösungen generiert.

Mithilfe der Typeinferenz werden die fehlenden Typen berechnet. Dafür greift diese auf die Informationen aus dem AST zu. Zusätzlich kennt diese den Additionsoperatur und weiß daher, für welche Typen dieser definiert ist.

Die erste Information aus dem AST ist der Typ des Parameters *a* in der zweiten Implementierung. Da der Parameter direkt zurückgegeben wird, muss der Rückgabetyp der Methode *Boolean* eine Superklasse von Boolean sein.

Für die erste Implementierung der Methode besitzt der Algorithmus keine direkte Information über den Typ des Parameters oder des Rückgabewertes. Dieser weiß nur, dass ein Wert als Parameter übergeben wird und dieser mithilfe des Plusoperators verarbeitet wird. Das Ergebnis

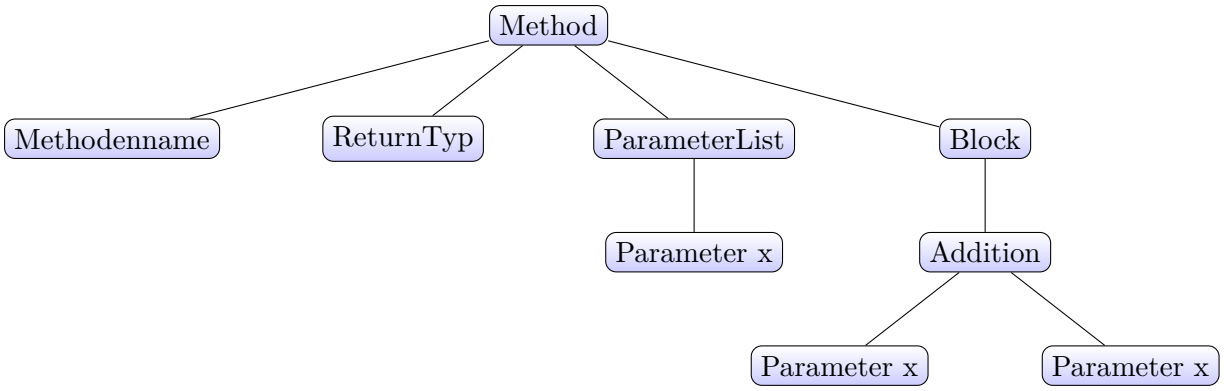

Abbildung 6.1: AST der Beispielmethode mit Addition

dieser Operation wird zurückgegeben. Da er jedoch weiß, für welche Typen dieser Operator definiert ist, kann er die Typen berechnen.

Die Lösungen der Typinferenz sind in Listing 6.2 dargestellt. Auf der linken Seite sind die Typen der Parameter und rechts vom Pfeil sind die Rückgabetypen dargestellt. Es wird deutlich, dass alle numerischen Typen als Eingabe verwendet werden können und dabei immer der Typ Number als Rückgabetyp verwendet wird. In diesen vier Fällen wird die Addition ausgeführt. Zusätzlich wurde der Typ String als Lösung gefunden. Dieser wird dabei auch als Rückgabetyp verwendet.

```
1 (Integer)     -> Number
2 (Long)        -> Number
3 (Float)       -> Number
4 (Double)      -> Number
5
6 (String)      -> String
```

Listing 6.2: Typinferenzlösungen der Klasse Beispiel

Mithilfe des AST, besonders dessen Platzerhaltern für die Typen, sowie den Lösungen der Typinferenz ist es möglich den Bytecode zu generieren. Es wird für jede Lösung des der Typinferenz eine Methode im Bytecode generiert, sodass diese überladen wird. Als Beispiel für den Bytecode wird die Addition von Integers in Listing 6.3 dargestellt. In der Signature und dem Deskriptor wird deutlich, dass der Eingabeparameter von Typ Integer und die Rückgabe von Typ Number ist.

Das Addieren zweier Zahlen erfolgt immer in der selben Struktur, es müssen nur jeweils andere Operatoren aufgerufen werden. Der erste Schritt ist das Laden der Zahlen. Dies wird durch den Aufruf von *aload_1*, sowie dem Aufruf der Methode *intValue* auf dem Integerobjekt, das man davor geladen hat. Dies muss man zwei mal ausführen, sodass die zwei Integerwerte auf dem Stack liegen. Im zweiten Schritt werden diese addiert. Dies erfolgt durch den Aufruf des *iadd*-Operators. Der letzte Schritt ist das Umwandeln des primitiven Datentypes *int* in die Klasse *Integer*. Dieser kann mithilfe von *areturn* zurückgegeben werden. Das Addieren von anderen numerischen Datentypen erfolgt analog. Das Konvertieren zu den primitiven Datentypen und zu den Wrapperklassen, sowie die eigentliche Addition müssen aber an den speziellen Typ angepasst werden.

```
1  public java.lang.Number method(java.lang.Integer);
2    descriptor: (Ljava/lang/Integer;)Ljava/lang/Number;
3    flags: ACC_PUBLIC
4    Signature: #18    // (Ljava/lang/Integer;)Ljava/lang/Number;
5    Code:
6      stack=2, locals=2, args_size=2
7        0: aload_1
8        1: invokevirtual #12    // Method java/lang/Integer.intValue:()I
9        4: aload_1
10       5: invokevirtual #12    // Method java/lang/Integer.intValue:()I
11       8: iadd
12       9: invokestatic  #16    // Method java/lang/Integer.valueOf:(I)Ljava/lang/
             Integer;
13       12: areturn
```

Listing 6.3: Bytecode der Integeraddition

Analog zu diesem Beispiel wird für jede Lösung der Typinferenz eine Methode im Bytecode generiert. Wichtig ist dabei, dass die Kombination aus Methodennamen und Parametertypen nicht doppelt vorhanden ist, da sonst die JRE die Ausführung mit einen Fehler beendet. Dies wird aber automatisch wie in Kapitel 5.3 beschrieben verhindert.

7 Zusammenfassung und Ausblick

In dieser Studienarbeit wurde die Generierung von Bytecode des Forschungsprojekt Java-TX weiterentwickelt. Zu Beginn dieser Arbeit war es möglich untypisiertes Java zu parsen, mithilfe von Typinterferez zu typisieren und Bytecode zu erzeugen.

Das Ziel war es, die Generierung von Bytecode erweitern, sodass alle Lösungen der Typinterferenz hinzugezogen werden und dadurch noch allgemeinere Klassen generiert werden. Als Problem standen dabei schon zu Beginn die generischen Datentypen fest. Diese werden im offiziellen Standard von Java durch *Type Erasure* gelöscht.

Durch die Einführung einer neuen Syntax der Signatur, konnte das Problem der Typlöschung umgangen werden. Auch ist es nun möglich, alle Lösungen der Typinterferenz zu verwenden. Dabei wird verhindert, dass eine Methode mit der selben Deskription mehrfach generiert wird, da sonst die JRE die Methoden nicht mehr unterscheiden kann und die Ausführung mit einem Fehler beendet.

Mit der Einführung der neuen Syntax konnte das Problem der Typlöschung jedoch nur teilweise gelöst werden. Es ist nun nicht mehr möglich die *Super* und *Extends*-Wildcard Typen zu verwenden. Außerdem treffen Probleme der Mehrfachvererbung auf.

Diese Probleme gilt es in zukünftigen Arbeiten zu lösen, sodass der Compiler keine Einschränkungen zum offiziellen Java mehr besitzt, sondern Java verbessert. Zusätzlich zu dem Beheben der Einschränkungen, sollte ein Algorithmus entwickelt werden, der aus der Menge von Typlösungen die zu generierenden Methoden intelligent auswählt.

Literatur

[Apa14] Apache Software Foundation. *Apache Commons BCEL*. 2014. URL: https://commons.apache.org/proper/commons-bcel/.

[Fik+15] Evelyn Fikus u. a. *Bytecode-Generierung eines neuartigen JavaCompilers*. 2015.

[Gos+15] James Gosling u. a. *The Java Language Specification*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[Lin+14] Tim Lindholm u. a. *The Java virtual machine specification: Java SE 8 edition*. Java SE 8 edition. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN: 013390590X.

[Ora16] Oracle, Hrsg. *Description of Java Conceptual Diagram*. 2016. URL: http://docs.oracle.com/javase/8/docs/index.html.

[Plü15a] Prof. Dr. Martin Plümicke. „Another Extension of the Java Type System“. In: (2015).

[Plü15b] Prof. Dr. Martin Plümicke. *Das Java-TX Projekt*. 2015. URL: http://www.hb.dhbw-stuttgart.de/~pl/JCC.html (besucht am 20. 10. 2015).

[Ste16] Florian Steurer. *Implementierung eines Typunifikationsalgorithmus für Java 8*. 2016.

[WSH12] Reinhard Wilhelm, Helmut Seidl und Sebastian Hack. *Übersetzerbau: Band 2: Syntaktische und semantische Analyse*. EXamen.press. Berlin, Heidelberg: Springer, 2012. ISBN: 3642011357.