Bytecode-Generierung eines neuartigen Java Compilers

Evelyn Fikus, Franziska Fütterling, Julia Schubert, Andreas Stadelmeier, Martin Plümicke

Duale Hochschule Baden-Württemberg,
Jägerstraße 56, 70174 Stuttgart, Deutschland
{evelyn.fikus,franziska.fuetterling,julia.schubert}@hp.com
a.stadelmeier@hb.dhbw-stuttgart.de
pl@dhbw.de
http://www.dhbw-stuttgart.de/home/

Zusammenfassung. In Java war es bisher nötig alle Typen von Methoden explizit anzugeben. Oftmals lassen sich Typen aber auch ohne direkte Angabe berechnen. Dazu haben wir ein System entwickelt, das bisher nur die Typen berechnen konnte und Java-ähnlichen Code erzeugt hat. Um den Code zu kompilieren, musste der Code so verändert werden, dass insbesondere die Klassenparameter verändert werden, was dazu führte, dass alle Deklarationen und new-Aufrufe ebenfalls angepasst werden mussten. Durch die nun eigenständige Bytecode-Generierung ist eine durchgängige Compilation in unserer Umgebung möglich.

1 Einleitung

Im Rahmen mehrerer Forschungsarbeiten wurde ein neuartiger Java Compiler entwickelt, der Typinferenz in Java ermöglicht[1]. Mithilfe von Algorithmen wird dabei der Typ eines gegebenen Ausdrucks rekonstruiert, sodass auf die explizite Angabe von Typen verzichtet werden kann[2].

Im Jahr 2013 wurde der von dem neuartigen Java Compiler eingesetzte Typinferenzalgorithmus erweitert und auf die neueste Java Version 8 angepasst. Ein zentrales Feature, das von Java 8 neu eingeführt wurde, stellen Lambda-Ausdrücke dar. Hierbei handelt es sich um ein Sprachkonstrukt, das von der funktionalen Programmierung inspiriert ist[3]. Dadurch wird in Java die Möglichkeit geschaffen, namenlose Methoden zu definieren und diese einer Variable zuzuweisen, die anschließend beispielsweise auch als Parameter an eine andere Methode übergeben werden kann[4].

Unter Verwendung des entwickelten, neuartigen Compilers wird angestrebt, die Schreibweise und die Syntax der Lambda Ausdrücke noch stärker an das funktionale Programmierparadigma, wie es zum Beispiel in Haskell vorzufinden ist [5], anzunähern. Dazu soll innerhalb der Lambda-Ausdrücke ebenfalls Typinferenz zum Einsatz kommen.

Nachfolgend wird ein Code-Beispiel dargestellt, das zunächst aufzeigt wie dessen Implementierung in Java 8 aussieht und anschließend demonstriert welche Vereinfachungen durch den neuartigen Compiler erreicht werden sollen:

Beispiel in Java 8:

```
class Bytecode<B> {
  Function<B, B> id = (B x) -> x;
}
```

Beispiel mit Vereinfachungen durch neuartigen Compiler:

```
class Bytecode {
  id = x -> x;
}
```

Diese Schwachpunkte werden durch den neuartigen Compiler und der darin mitgelieferten Typinferenz beseitigt. Wie in der zweiten Umsetzung zu sehen ist, entfällt bei dessen Einsatz die explizite Angabe der generischen Typen. Bei der Verwendung des Lambda-Ausdrucks wird durch den Typinferenzalgorithmus automatisch der korrekte Typ inferiert und eingesetzt. Dabei wird zusätzlich ermöglicht, dass für jeden Ausdruck individuell ein variabler Typ vergeben werden kann. Der von der Typinferenz erzeugte Code sieht demnach folgendermaßen aus:

```
class Bytecode {
  <B> Function<B,B> id = (B x) -> x;
}
```

Die zugrundeliegende Motivation ist es, das in Haskell charakteristische Konstrukt der Typvariablen zur Definition von Funktionen, die für alle möglichen Variablentypen gültig sind, in Java abzubilden. Es soll somit realisiert werden, dass stets mit einem möglichst allgemeinen Typen gearbeitet wird. Dadurch wird schließlich eine dynamische Handhabung von variablen Typen in Java ermöglicht.

Ein erster Ansatz wäre den nach Typinferenz erzeugten Java Code durch den Standard Java-Compiler zu übersetzen. Da der Standard-Java aber keine generischen Typvariabeln für Attribute kennt, ist auf alle Fälle eine Code-Transformation nötig. Man könnte die Typvariablen als generische Typen der Klasse definieren, was den Nachteil hätte, dass alle new-Aufrufe angepasst werden müssten. Oder man könnte die generischen Typvaraibeln durch Object ersetzen. Wir haben uns für die direkte Erzeugung von Bytecode entschieden, auch weil zukünftige Erweiterungen, dann einfacher zu realisieren sind.

2 Ähnliche Arbeiten

Es gibt verschiedene Bibliotheken, die der Analyse, Manipulation und Erzeugung von Bytecode dienen. Für die Generierung des Bytecodes innerhalb des neuartigen Compilers wurde die Nutzung unterschiedlicher Bibliotheken abgewogen.

Eine der in Erwägung gezogenen Bibliotheken ist Jasmin[6]. Allerdings hat sich herausgestellt, dass diese Bibliothek nicht für die Erzeugung des Bytecodes geeignet ist. Die Ursache liegt darin, dass der zu übersetzende Code in der

Jasmin spezifischen Syntax vorliegen müsste, aus der schließlich der Bytecode erzeugt wird. Da der Code innerhalb des neuartigen Compilers nicht in der Jasmin Syntax vorliegt, konnte diese Bibliothek nicht verwendet werden.

Außerdem wurde ebenfalls das Framework ASM in Betracht gezogen, um die Erzeugung des Bytecodes innerhalb des neuartigen Compilers zu realisieren[7]. ASM zeichnet sich dadurch aus, dass es besonders schnell beim Parsen des Bytecodes ist[8]. Die Nutzung dieses Frameworks ist darauf ausgerichtet, dass auf einer tiefen Ebene ohne Abstrahierungsgrad gearbeitet wird. Dafür stellt ASM viele Funktionen bereit.

Eine weitere Bibliothek, die für die angestrebte Generierung von Bytecode in Frage kommt, ist BCEL[9]. Charakteristisch für BCEL ist, dass vorgefertigte Konstrukte mitgeliefert werden, die von der konkreten Erzeugung des Bytecodes abstrahieren. Ein Beispiel dafür stellt die InstructionFactory dar[10]. Dadurch muss der Entwickler nicht auf der untersten Ebene arbeiten und der Einstieg in die Nutzung der Bibliothek wird somit vereinfacht.

Schlussendlich wurde die Bibliothek BCEL für die Erzeugung des Bytecodes im neuartigen Java Compiler ausgewählt, da die Geschwindigkeitsvorteile, die ASM bieten würde, in diesem Fall nicht die höchste Priorität haben. Ausschlaggebend waren vielmehr die von BCEL zur Verfügung gestellten abstrakten Konstrukte, die die Bytecode-Generierung insgesamt einfacher gestalten. Dadurch sollte eine schnelle und erfolgreiche Einarbeitung begünstigt werden.

3 Bytecode-Generierung mit BCEL

In Java wird Quellcode mit Hilfe des Compilers in binäre Klassendateien umgewandelt. Diese Klassendateien werden auch als Bytecode bezeichnet. Oftmals soll in bereits bestehende Javaprogramme eingegriffen werden, um beispielsweise eine Performanzsteigerung herbeizuführen oder Java parametrisierte Typen hinzuzufügen. Um diese Aspekte zu realisieren, muss der bestehende Java Compiler oder die Java Virtual Machine verändert werden. Ein viel effizienterer und plattformunabhängiger Weg ist, den vom Compiler erzeugten Bytecode manuell anzupassen. Dies ist ein Weg, den viele Informatiker gehen. Allerdings sind deren Lösungen oft projektabhängig und selten wiederverwendbar.

Die Byte Code Engineering Library (BCEL)[9] ist ein von der Apache Organisation gesponsertes Projekt. BCEL wird von Apache selbst als Werkzeugsatz bezeichnet, welcher Klassen und Schnittstellen zur statischen Analyse und Veränderung von Java Bytecode zur Verfügung stellt.

BCEL stellt drei Pakete zur Verfügung: Eines, welches Klassen beinhaltet, die statische Einschränkungen der Klassendateien realisieren und nicht für Bytecode Modifikationen gedacht sind. Eines, welches erlaubt, dynamisch JavaClass- oder Methodenobjekte zu generieren und modifizieren. Und das letzte, welches diverse Code Beispiele und Werkzeuge, beispielsweise zum Anzeigen von Klassendateien, bereitstellt. Die für dieses Projekt vorrangig verwendeten Teile der Bibliothek sind ClassGen, MethodGen und diverse InstructionHandles. Sie bilden die Abstraktionsebene zwischen Abstrakter Syntax und dem endgültigen

Bytecode. Sie speichern alle zur Bytecodegenerierung benötigten Informationen. Die Eintragung dieser Informationen in eine Classfile übernimmt im Anschluss das Framework. So kann BCEL beispielsweise den Konstantenpool eigentständig aus den gegebenen Informationen generieren. Im Folgenden werden die verwendeten Klassen genauer betrachtet:

– ClassGen liefert eine abstrakte Sicht für die dynamische Kreation und Transformation von Klassendateien. Hauptsächlich ist ClassGen für die Einhaltung der Beschränkungen von Java, wie die fest gecodeten *generic* Bytecode Adressen, zuständig. Auch MethodGen und ConstantPoolGen für die Realisierung von jeweils Methoden und Konstantenpool, sind in diesem Teil der Bibliothek zu finden. Folgende Abbildung 1 zeigt ein UML Diagramm des ClassGen der Bibliothek.

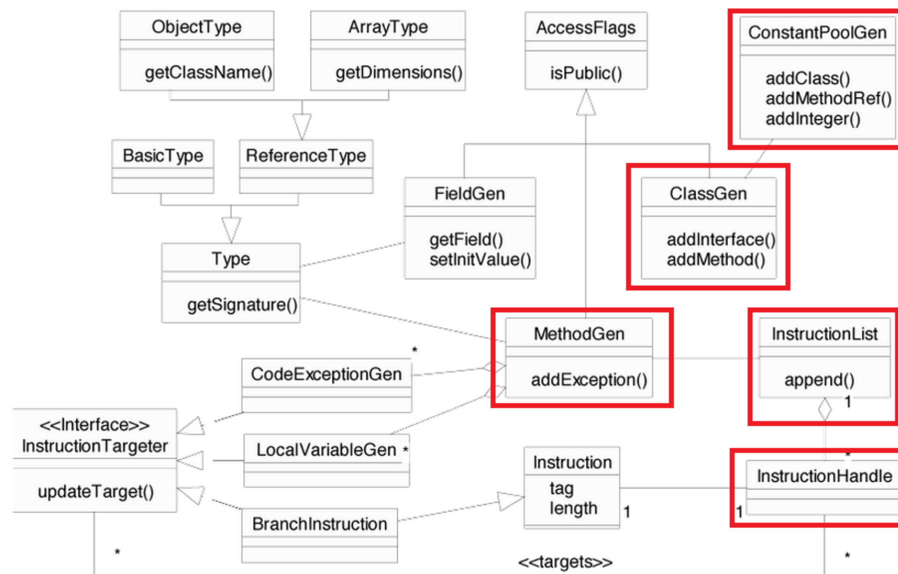

Abb. 1. UML of ClassGen[9]

– MethodGen wird verwendet, um Methoden der zu kompilierenden Klasse dem ClassGen hinzuzufügen. Des Weiteren vereint MethodGen alle vorhandenen InstructionHandles zu einer InstructionList.
– Viele der InstructionHandles stammen ursprünglich aus der InstructionFactory. Dieser Teil der Bibliothek soll die Generierung von Variablen, unter anderem von Konstanten, vereinfachen, indem bestimmte Instruktionen bereits vorgegeben werden. Die InstructionHandles an sich bilden den in diesem Projekt kleinsten Teil der BCEL Bibliothek. Sie werden für die Generierung von Bytecode für lokale sowie globale Variablen verwendet und schreiben

entsprechende Variablen mit Befehlen wie beispielsweise *iconst_1 (schreibt die Konstante 1)* in den Konstantenpool.

4 Vorgehensweise mit Hilfe von BCEL

Im Folgenden soll nun die Implementierung einer Methode zur Bytecode-Generierung innerhalb des neuartigen Java Compilers betrachtet und erläutert werden.

Im Syntaxbaum des neuartigen Compilers ist eine ähnliche Struktur zu finden, welches eine analoge Übertragung und Implementierung von BCEL ermöglicht. Zunächst wird die übergreifende Klasse Class.java des Compilers bearbeitet. Auf der gleichen Ebene in BCEL findet sich das ClassGen, welches innerhalb dieser Klasse den Hauptbestandteil für die Bytecode-Generierung bildet. Es wird zunächst also eine *genByteCode*- Methode implementiert, welche in der Lage ist, das ClassGen in eine Klassendatei zu schreiben. Dafür wird ein neues ClassGen deklariert, welches die dafür typischen Parameter, wie im vorigen Kapitel beschrieben, enthält. Diese Parameter sind dynamisch gestaltet, sodass bei Nutzung des Compilers die korrekten Daten des Programms des Anwenders in die Paramter eingetragen werden. Des Weiteren wird ein Konstantenpool erzeugt, welcher mit den Daten aus ClassGen gefüllt wird. Das ClassGen selbst wird an die unteren Hierarchieebenen des Compilers weitergegeben. Innerhalb dieses Compilers besteht die nächste Hierarchieebene aus Feldern. Entsprechend werden diese iteriert, sodass das ClassGen in allen Feldern gefüllt wird und wieder an die Class.java übergeben wird, welche als Ergebnis eine komplette Klassendatei zurückgibt. Nachfolgende Abbildung 2 soll den Zusammenhang zwischen den im Folgenden erläuterten Klassen veranschaulichen.

Die Felder, die innerhalb dieses Projektes die interessantesten sind, sind Methoden, umgesetzt in Method.java, beziehungsweise Konstruktoren. Der Parser des neuartigen Compilers unterscheidet nicht zwischen Methode und Konstruktor. Aus diesem Grund wird eine Methode, sollte sie die Kriterien eines Konstruktors erfüllen, nach dem Parsen in Constructor.java zu einem Konstruktor umgewandelt. Constructor.java erbt entsprechend von Method.java. Also muss in Method.java nun eine *genByteCode*-Methode implementiert werden. Die *genByteCode*-Methode in Method.java sieht wie folgt aus: Als Übergabeparameter wird die ClassGen übergeben. Dieses ClassGen ist das in Class.java erstellte, welches durch Method.java gefüllt wird. Der Konstantenpool des ClassGen wird lokal gespeichert. Es werden eine InstructionFactory, welche das ClassGen und den Konstantenpool übergeben bekommt, und eine InstructionList erzeugt.

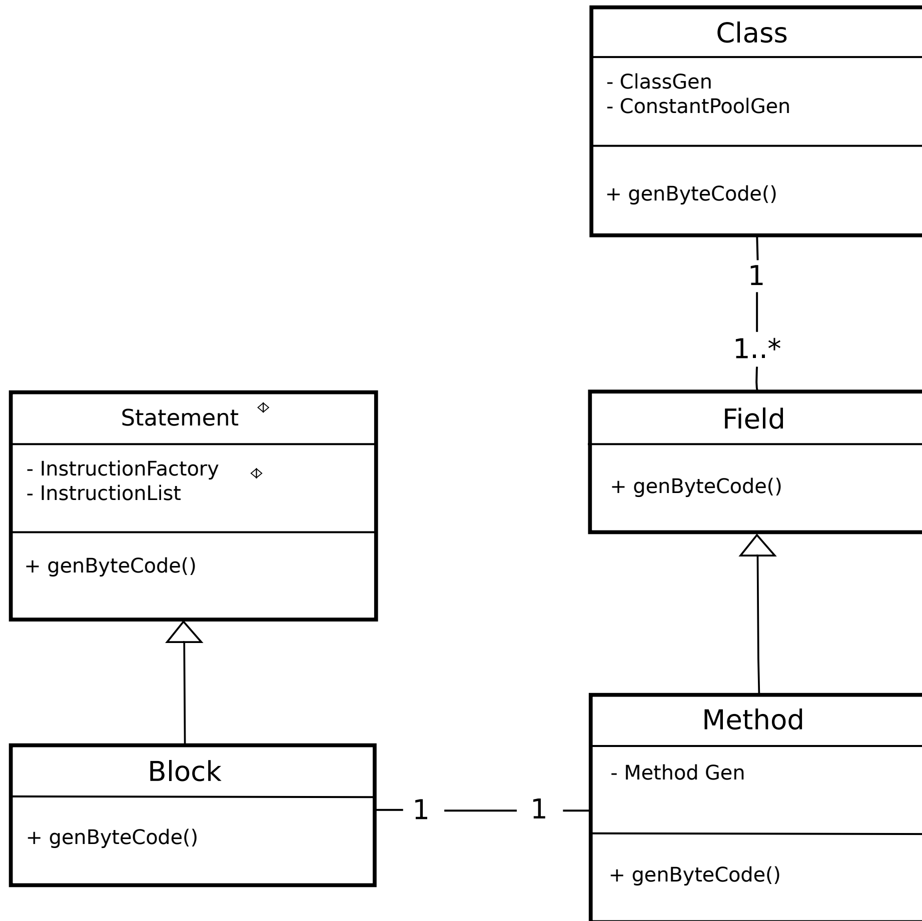

Abb. 2. Klassendiagramm

Diese werden im Verlauf der Methode verwendet beziehungsweise gefüllt. Anhand eines Objektes der Klasse Class.java kann im weiteren Verlauf der Name der Klasse, zu welcher die zu kompilierende Methode gehört, ausgelesen werden. Dies geschieht innerhalb der Erzeugung eines MethodGen. MethodGen braucht als Parameter die Access Flags, den Rückgabewert der Methode, deren Argumente, den Namen der Methode, den Namen der Klasse, zu welcher die Methode gehört, die InstructionList und den Konstantenpool. Eine Methode besitzt immer mindestens einen Block, welcher durch geschweifte Klammern gekennzeichnet ist. Der zur Methode zugehörige Block wird im nächsten Schritt in eine lokale Variable des Typs Block gespeichert. Anhand dieses Blocks kann die InstructionsList gefüllt werden, indem die *genByteCode*-Methode auf dem Block aufgerufen wird und das ClassGen dabei übergeben. Die resultierende InstructionList wird nun der Methode angefügt mit Hilfe des *append*-Befehls. Außerdem

wird geprüft, ob der Block überhaupt irgendwelche Statements enthält. Sollte dies nicht der Fall sein, oder aber das letzte verfügbare Statement keine Instanz des Return-Typs besitzen, wird hier manuell ein void (leer) zurückgegeben und an der InstructionList angefügt. Anschließend wird noch ein dynamischer Stack für diese Methode gesetzt, welcher seine Größe durch die InstructionList bestimmt. Zuletzt wird die Methode dem ClassGen hinzugefügt. Folgender Code Abschnitt zeigt die eben erläuterte Implementierung zunächst in Class.java und anschließend in Method.java.

```
1  /* genByteCode in Class.java
2  */
3  private InstructionFactory _factory;
4  private ConstantPoolGen    _cp;
5  private ClassGen           _cg;
6
7      public ByteCodeResult genByteCode() throws IOException {
8
9        _cg = new ClassGen(name, superClass.get_Name(), name + ".java",
             Constants.ACC_PUBLIC , new String[] {  });
10       _cp = _cg.getConstantPool();
11       _factory = new InstructionFactory(_cg, _cp);
12
13       for(Field field : this.fielddecl){
14           field.genByteCode(_cg);
15         }
16
17       ByteCodeResult code = new ByteCodeResult(_cg);
18     return code;
19     }
20 /* genByteCode in Method.java
21 */
22 @Override
23   public void genByteCode(ClassGen cg) {
24     ConstantPoolGen _cp = cg.getConstantPool();
25     InstructionFactory _factory = new InstructionFactory(cg, _cp);
26     InstructionList il = new InstructionList();
27     Class parentClass = this.getParentClass();
28
29     MethodGen method = new MethodGen(Constants.ACC_PUBLIC, this.getType().
           getBytecodeType(), org.apache.bcel.generic.Type.NO_ARGS , new String
           [] {  }, this.get_Method_Name(), parentClass.name, il, _cp);
30
31     Block block = this.get_Block();
32     InstructionList blockInstructions = block.genByteCode(cg);
33
34     il.append(blockInstructions);//Die vom Block generierten Instructions an die
           InstructionList der Methode anfügen
35
36     if (block.get_Statement().size() == 0) { il.append(_factory.createReturn(
           org.apache.bcel.generic.Type.VOID)); }
37     else {
38         if (!(block.get_Statement().lastElement() instanceof Return)) { il.
               append(_factory.createReturn(org.apache.bcel.generic.Type.VOID));
                }
39     }
40
41     method.setMaxStack(); //Die Stack Größe automatisch berechnen lassen (erst nach dem
           alle Instructions angehängt wurden)
42
43     cg.addMethod(method.getMethod());
44
45   }
```

Nun wurden bereits zwei höhere Hierarchieebenen des Compilers betrachtet. Der bereits erwähnte Block bildet die nächst niedrigere Hierarchieebene. Blöcke werden innerhalb des Compilers in Block.java umgesetzt. Blöcke sind selbstständig nicht existenzfähig, da leere geschweifte Klammern in Java keine Bedeutung besitzen. Entsprechend müssen diese gefüllt werden - Mit Hilfe von Ausdrücken, welche im Compiler durch Statements ausgedrückt sind. Block erbt von Statement und besitzt eine eigene *genByteCode*-Methode, welche eine InstructionList zurückgibt, die durch die Iteration über Statement gefüllt wird. Statement bildet den in diesem Projekt kleinsten, bearbeiteten Bestandteil des Compilers und damit auch die niedrigste, bearbeitete Hierarchieebene.

Damit die Abläufe der Interaktionen zwischen den in Abbildung 2 dargestellten Klassen klar werden, stellt nachfolgende Abbildung 3 ein Sequenzdiagramm dieser Klassen zur Verfügung.

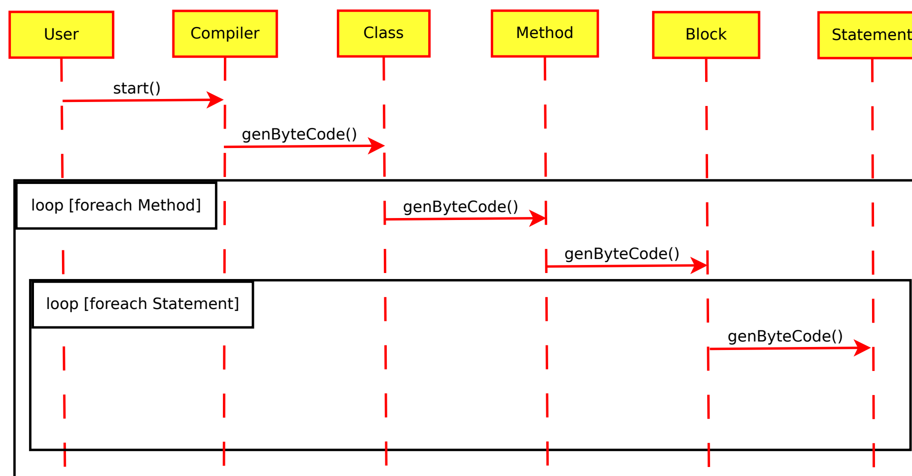

Abb. 3. Sequenzdiagramm

5 Fazit und Ausblick

Zum Zeitpunkt dieser Arbeit lässt sich abschließend sagen, dass ein Grundgerüst zur Vervollständigung und Erweiterung der Bytecode-Generierung erstellt wurde. Dieses Grundgerüst beinhaltet die Existenz einer Methode zur Bytecode-Generierung in allen dazu notwendigen Klassen. Aufgrund der zeitlichen Begrenzung des vorliegenden Projektes sowie der Unvollständigkeit der Parser-und Typinferenz-Funktionalitäten, auf die die Bytecode-Generierung aufbaut, sind nicht alle dieser Methoden funktionsfähig.

Zukünftige Projekte können auf dieser Arbeit aufbauen. Anpassungen am ausgegebenen Bytecode des Compilers sind nun leicht möglich. Zuvor konnte

der neuartige Compiler zwar Typen im Java Quellcode inferieren und einsetzen, aber nicht eigentständig Bytecode erzeugen. Diesen Schritt musste ein Standard Java Compiler übernehmen wodurch kein Einfluss auf das Kompilat möglich war.

Dies ermöglicht unter anderem die Einführung von echten Generischen Typen in den Bytecode [13].

Quellenverzeichnis

1. Stadelmeier, A. and Plümicke, M., *Implementierung eines Typinferenzalgorithmus für Java 8*, in *Tagungsband des 17. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'13)*, Ausgabe 2014/02, Seiten 127–134, *http://www.ba-horb.de/~pl/papers/Typinferenz-Java8.pdf*, University Halle-Wittenberg, Institute of Computer Science 2014

2. Ehlers, C., *Typinferenz*, *http://www.fh-wedel.de/~si/seminare/ws04/Ausarbeitung/4.Typcheck/staTyp4.htm*, o.J.

3. Inden, M., *Buch Java 8 - Die Neuerungen*, *Vgl. S. 3*, dpunkt.verlag GmbH 2014

4. Stadelmeier, A., *Java type inference as an Eclipse plugin*, in *Proceedings of the Studierendenkonferenz Informatik 2015*, 2015

5. Inden, M., *Buch Java 8 - Die Neuerungen*, *Vgl. S. 174*, dpunkt.verlag GmbH 2014

6. Meyer, J., *Jasmin Home Page*, *http://jasmin.sourceforge.net/*, Jasmin Home Page, 2004

7. OW2 Consortium, *ASM*, *http://asm.ow2.org/*, ASM, 2015

8. Bruneton, E. and Lenglet, R. and Coupaye, T., *ASM: a code manipulation tool to implement adaptable systems*, *http://asm.ow2.org/current/asm-eng.pdf*, o.J.

9. Apache Foundation, *Apache Commons BCEL*, *https://commons.apache.org/proper/commons-bcel/manual.html*, 2014.

10. The Apache Software Foundation, *Class InstructionFactory*, *http://commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/generic/InstructionFactory.html*, The Apache Software Foundation, 2014

11. Lindholm, T. and Yellin, F. and Bracha, G. and Buckley, A., *The Java® Virtual Machine Specification*, *Java SE 8*, Addison-Wesley 2014

12. o.A., *JDK 8*, *http://openjdk.java.net/projects/jdk8/*, OpenJDK, 2014

13. Plümicke, M., *Java Type System – Proposals for Java 10 or 11*, to appear in *Tagungsband zum 19. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015