Implementierung eines Typinferenzalgorithmus für Java 8

Andreas Stadelmeier und Martin Plümicke

1. November 2013

Abstract

Java 8, die nächste anstehende Erweiterung der Programmiersprache Java, führt neue Features in der Sprache ein. Dabei handelt es sich unter anderem um sogenannte Lambda-Ausdrücke, welche das Ablegen von Funktionen in Variablen ermöglichen und dadurch Elemente der funktionalen Programmierung in Java einführen.
Dieser Artikel beschreibt die Implementierung eines zu Java 8 kompatiblen Typinferenzalgorithmus. Standardmäßig ist Java streng typisiert. Jeder Variablen muss bei der Deklaration ein Typ zugewiesen werden. Ein Typinferenzalgorithmus kann diesen Arbeitsschritt jedoch übernehmen, indem er den Typ einer Variablen anhand ihrer Verwendung ermittelt und einsetzt.

1 Einleitung

1.1 Problemstellung

In der statisch typisierten Sprache Java müssen die Typen sämtlicher Variablen und Ausdrücke bei deren Dekleration festgelegt werden. Dabei lässt sich auf deren Datentypen oft auch anhand ihres Kontexts schließen. Knapp 20 Studenten entwickelten im Zuge mehrerer Studienarbeiten (z.B. [Bäu05, Hol06, Lüd07]) einen Java-Compiler, welcher in der Lage war selbständig Typen für Java 7 zu inferieren.
Mit Java 8 erhält die Programmiersprache neue Features, welche erstmalig Elemente der funktionalen Programmierung in Java einführen. Diese sollen unter anderem die Verwendung von inneren Klassen teilweise ablösen. Sogenannte Lambda Ausdrücke, eine kompakte Form der Methodendeklaration, ermöglichen zukünftig die Übergabe von Funktionen als Parameter bei Methodenaufrufen oder das Speichern von Funktionen in Variablen. Allerdings unterstützt der ursprünglich implementierte Typinferenzalgorithmus die neuen Sprachkonstrukte nicht.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist die Anpassung des Java 7 Compilers an die neuen Features in Java 8 und die Ersetzung dessen Typinferenzalgorithmus. Dieser Compiler ist auf die in Java 7 verwendbaren Datentypen ausgelegt. Weder der Compiler noch sein Typinferenzalgorithmus sind in der Lage Lambda Ausdrücke zu verarbeiten. Der erste Schritt ist es daher den Parser an die mit Java 8 eingeführten Änderungen anzupassen. Anschließend kann die überarbeitete und zu Lambda Ausdrücken kompatible Spezifikation des Typinferenzalgorithmus implementiert werden.

1.3 Ziel

Ziel dieser Arbeit ist die Umsetzung eines zu Java 8 kompatiblen Compilers mit dem in [Plüar] spezifizierten Typinferenzalgorithmus. Der Aufbau der Arbeit teilt sich in vier Teile ein. Im folgenden Kapitel 2 sind notwendigen Grundlagen und Informationen hauptsächlich zu den Neuerungen in Java 8 erklärt. In Kapitel 3 die Typen von Lambda Ausdrücken näher betrachtet. Die Analyse des bereits vorhandenen Java-Compilers und der Datentypen, welche in dieser Arbeit wiederverwendet werden, befindet sich in Kapitel 4. Anschließend ist die Konkrete Umsetzung des neuen Typinferenzalgorithmus in Kapitel 5.3 beschrieben. Im letzten Abschnitt werden die erreichten Ergebnisse und das geplante weitere vorgehen vorgestellt.

2 Grundlagen

2.1 Java-Compiler

Ein Compiler ist ein Programm das in der Lage ist eine Programmiersprache in eine andere Sprache zu übersetzen. Die meisten Compiler wandeln den für Menschen leichter verständlichen Quellcode in eine für Rechner ausführbare Sequenz von Maschinenbefehlen um. Maschinencode ist dabei meist plattformspezifisch, kann also nur von einem bestimmten Betriebssystem und einer bestimmten Prozessorarchitektur ausgeführt werden. Ein Java-Compiler hingegen übersetzt die Programmiersprache Java nicht direkt in Maschinencode, sondern in einen von der Java-Laufzeitumgebung interpretierbaren Bytecode. Die Java-Laufzeitumgebung ist ein Interpreter für Java-Bytecode, welcher für eine vielzahl von Architekturen angeboten wird. Anstatt den Bytecode in eine andere Sprache zu übersetzen interpretiert die virtuelle Maschine ihn zur Laufzeit und führt die einzelnen Anweisungen nacheinander aus. Dank dieser Vorgehensweise ist ein kompiliertes Java-Programm auf allen unterstützten Plattformen ausführbar. [Her11]

2.2 Lambda-Ausdrücke

Ein Lambda Ausdruck ist ein mit Java 8 neu eingeführtes Sprachkonstrukt. Es erlaubt die Erstellung von anonymen Funktionen innerhalb einer Methode oder Klasse. Ein Lambda Ausdruck darf an allen Stellen zum Einsatz kommen an denen ein Ausdruck erlaubt ist. Er kann also z.B. als Parameter einer Funktion übergeben werden, als Rückgabetyp einer Methode dienen oder einer Variable zugewiesen werden.
Ein Lambda Ausdruck besteht aus zwei Teilen. Auf der linken Seite steht eine Parameterliste, welche Parametertyp und Parameternamen der Funktion auflistet. Auf der rechten Seite befindet sich der eigentliche Funktionsrumpf. Der Rückgabetyp muss nicht angegeben werden. [OP10]

Listing 1: Beispielhafte Benutzung eines Lambda Ausdrucks in einer foreach-Schleife

```
1 interface OneArg<A> {
2   void invoke(A arg);
3 }
4
5 <T> void forEach(Collection<T> c, OneArg<T> block){
6   for (Iterator<T> it = c.iterator(); c.hasNext();){
7     block.invoke(it.next());
8   }
9 }
```

2.3 Funktionales Interface

Ein funktionales Interface ist ein Interface mit nur einer einzigen abstrakten Methode. Ein solches Interface wird auch als SAM-Typ - “Single Abstract Method - Type“ bezeichnet. In der Java Standardbibliothek sind bereits einige SAM-Typ Schnittstellen vorhanden, wie zum Beispiel “Runnable“, “Callable“ oder “EventHandler“. Zusätzlich zu seiner einzigen abstrakten Methode darf ein funktionales Interface noch weitere Methoden besitzen, solange diese nicht als abstrakt definiert sind. Diese Ausnahme ist notwendig, da jedes Objekt und Interface in Java per Definition von der Klasse Object abgeleitet ist und dadurch dessen Methoden erbt.
In Verbindung mit einem Typinferenzalgorithmus spielen funktionale Interface eine besondere Rolle, da ein Lambda Ausdruck als Typ nur ein funktionales Interface annehmen kann. [Naf13]

3 Typ eines Lambda Ausdrucks

Von zentraler Bedeutung für die Konzeption eines Typinferenzalgorithmus für Lambda Ausdrücke ist die Kenntnis über deren Typisierung. Ein Lambda Ausdruck ist bei seiner Definition keinem bestimmten Interface zugeordnet. Je nachdem in welchem Kontext er eingesetzt wird ergibt sich sein Typ. Ein Lambda Ausdruck kann also mehrfach eingesetzt werden und dabei die Typen verschiedener Schnittstellen annehmen. Ein Ausdruck ist zum Typ eines Interface kompatibel, wenn folgende Bedingungen erfüllt sind:

1. Es muss sich um ein funktionales Interface handeln
2. Die Anzahl und Datentypen der Parameter des Lambda Ausdrucks müssen mit denen der SAM-Funktion der Schnittstelle übereinstimmen
3. Die Ausdrücke, welche vom Lambda Ausdruck zurückgegeben werden müssen kompatibel mit dem Rückgabetyp der SAM-Funktion sein.

Wird ein Lambda Ausdruck einer Variable zugewiesen kann diese in alle kompatiblen Interfacetypen gecastet werden.

3.1 Typzuweisung durch den Typinferenzalgorithmus

Bei der Spezifikation des Typinferenzalgoritmus [Plüar] werden unendlich viele funktionale Interfaces definiert. Mit ihnen ist es möglich die Typen aller möglichen Lambda Ausdrücke darzustellen. Diese Interface sind folgendermaßen aufgebaut:

```
1 interface FunN<R, T1, T2, ... , TN> {
2         R apply(T1, ... TN);
3 }
```

Jeder Lambda Ausdruck verfügt über einen Rückgabetyp R und N (T1 bis TN) Parameter, welche wiederum jeweils einen Datentyp besitzen. Jedes dieser generischen Interface-Definitionen spezifiziert eine abstrakte Methode namens “apply“. Diese Methode besitzt den Rückgabetyp ’R’ und ’N’ Parameter, welche die Datentypen ’T1’-’TN’ besitzen. Durch korrektes setzen der generischen Typen dieser Interfaces kann zu jedem Lambda Ausdruck ein passendes Funktionales Interface dargestellt werden.

Listing 2: Beispiel

```
1 Fun1<String, String> (string par1) -> {return par1+par1;}
```

3.2 Benutzung von Variablen innerhalb eines Lambda Ausdrucks

Innerhalb eines Lambda Ausdrucks können lokale Variablen deklariert werden. Für deren Gültigkeitsbereich gelten dabei die üblichen Regeln, wodurch diese nur innerhalb der durch den Lambda Ausdruck definierten Funktion gelten. Zusätzlich kann ein Lambda Ausdruck auf die Instanz der Klasse, in welcher dieser deklariert wurde, sowie deren Felder und Methoden zugreifen. Selbst der Zugriff auf lokale Variablen ist unter der Einschränkung, dass diese als "final"deklariert wurden, möglich. Hinweis: Durch eine Neuerung in Java 8 gelten lokale Variablen auch ohne explizite "finalDefinition als "final", sofern diese nur bei ihrer initialisierung beschrieben werden.
Daraus folgt, dass auch Instruktionen innerhalb des Lambda-Ausdrucks für den Typinferenzalgorithmus von Bedeutung sind. Nicht nur zur Inferierung der lokalen Variablen und Parameter der Lambda-Funktion. Auch auf die Typen der Felder und Variablen der umschließenden Klasse kann geschlussfolgert werden, sofern diese im Lambda Ausdruck Verwendung finden.

4 Wiederverwendung des alten Typinferenzalgorithmus

Die Implementierung eines Typinferenzalgorithmus in den Compiler fand bereits 2006 während der Studienarbeit [Bäu05] statt. Eine neue Spezifikation von Martin Plümicke [Plüar], welche im Zuge dieser Studienarbeit umgesetzt wird, ersetzt den alten Algorithmus. Der neue Typinferenzalgorithmus kann aber einige Datentypen der alten Implementierung wiederverwenden. Im folgenden werden die übernommenen Klassen und Methoden aus [Bäu05] vorgestellt:

Pair Die Klasse Pair verknüpft zwei Objektreferenzen miteinander. Sie wurde in der Projektarbeit [Bäu05] zur Speicherung von Typabbildungen eingesetzt. Diese Klasse findet auch im neuen Typinferenzalgorithmus Verwendung. Sie kann eingesetzt werden um Zusammenhänge zweier Objekte darzustellen, was zum Beispiel bei Typabbildungen oder Constraints der Fall ist.

TypeAssumption Die Klasse TypeAssumption stellte im Vorgängerprojekt eine Typannahme dar. Auch der neue Algorithmus arbeitet mit Mengen von Typannahmen, wofür er diese Datenstruktur des Vorgängerprojekts benutzt.

CSet<E> Während der Studienarbeit [Bäu05] wurde bei der Umsetzung des Typinferenzalgorithmus auch eine Datenstruktur zur Speicherung, Vereinigung und Abbildung von Mengen implementiert. Dies war notwendig, da die Java Standardbibliothek keine entsprechende Datenstruktur anbietet. Die Klasse "CSet<E>ïst generisch, kann deshalb jeden beliebigen Datentyp enthalten und bietet sich besonders für eine Wiederverwendung an. Diese Klasse kommt in der Implementierung des neuen Typinferenzalgorithmus unter anderem bei der Speicherung von Typannahmen und Typconstraints (siehe Kapitel 5.2) zum Einsatz.

TypePlaceholder Die Klasse “TypePlaceholder“ ist eine Unterklasse von “Type“ und kann daher im Syntaxbaum als Platzhalter an die Stelle eines Typs gesetzt werden. Nach dem Parsen eines Java-Quellcodes durch den Compiler bleiben die Typangaben zu Elementen im Syntaxbaum leer, zu denen zu diesem Zeitpunkt noch keine Typen bekannt sind. An diese Stellen platziert der Typinferenzalgorithmus TypePlaceholder. Jede Instanz eines TypePlaceholders speichert eine Referenz auf die Variablen und Ausdrücke an denen sie als Typ platziert wurde. Der TypePlaceholder dient also als Observer an dem sich mehrere Subjekte registrieren. Dies erleichtert das Einsetzen der mittels des Typinferenzalgorithmus errechneten Typen an die Stellen der Platzhalter. Hat der Algorithmus den Typ eines Platzhalters ermittelt, kann durch aufrufen einer Methode der entsprechenden TypePlaceholder-Instanz dieser Typ an die am Placeholder registrierten Ausdrücke, Variablen und Anweisungen übermittelt werden. Diese ersetzen den TypePlaceholder durch den berechneten Typ. Dadurch kann ein Typ mit nur einem einzigen Aufruf an alle betroffenen Stellen im Syntaxbaum eingesetzt werden.

5 Typinferenzalgorithmus

5.1 Ablauf des Typinferenzalgorithmus

Im folgenden ist der Ablauf des Typinferenzalgorithmus (Abb. 1) beschrieben.

1. Der JavaParser generiert die Klasse “SourceFile“. Sie speichert alle eingelesenen Klassen als “Class“-Objekte, welche jeweils einen abstrakten Syntaxbaum einer Java-Klasse darstellen mit dem der Typinferenzalgorithmus **TI** aufgerufen wird.

2. Die TYPE-Methode erstellt einerseits die benötigten Typ-Annahmen (TypeAssumptions). Dabei handelt es sich um die Felder *fdecls*$_t$ und Methoden *mdecls*$_t$ der geparsten Klassen. Dabei spielt es keine Rolle ob deren Typen bereits bekannt sind. Fehlende Typangaben werden mit TypePlaceholdern ersetzt. Sie werden in einer für den Typinferenzalgorithmus verfügbare Menge von Typannahmen gespeichert. Anschließend ruft er die TYPE-Methode, welche sich in der selben Klasse befindet, auf.

3. Die TYPE-Methode berechnet andererseits die Constraints ($ConS$) der Felder und Methoden des in Schritt 1 erstellten Syntaxbaumes.

4. Ein Unify-Algorithmus [Plü09] unifiziert die gesammelten Constraints. Dabei entsteht eine Lösungsmenge $\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}$ bestehend aus einer Restmenge von Constraints cs_i, an die keine Bedingung geknüpft ist und einer substitution σ_i als Lösung der unbekannten Typen.

5.2 Berechnen der Constraints

Den Hauptteil des Typinferenzalgorithmus bildet die Erstellung der Constraints. Ein Constraint ist eine Regel, die zwei Typen (darunter fallen auch TypePlaceholder) miteinander in Relation setzt. Der im Zuge dieser Arbeit implementierte Typinferenzalgorithmus erstellt Constraints der Form: “Typ A $<.$ Typ B“. Sie legen dabei die Einschränkung fest, dass “Typ A“ von “Typ B“ erben muss.
Die Erstellung der Constraints beginnt mit dem Aufruf der TYPE-Methode in der Class-Klasse

(siehe Kapitel 5.1). Von dieser Methode aus durchläuft eine Tiefensuche den bereits generierten Syntaxbaum des zu inferierenden Java-Quellcodes. Im Endeffekt werden alle Anweisungen und die darin befindlichen Ausdrücke der Klasse nacheinander abgearbeitet. Auf ihnen ruft der Algorithmus die Methoden TYPEStmt (für Anweisungen) oder TYPEExpr (für Ausdrücke) auf. Die Methoden haben Zugriff auf das bisher erstellte ConstraintsSet und die bisher vorhandenen Assumptions. Sie fügen auf Basis der vorhandenen Assumptions neue Constraints hinzu. Hat TYPE den gesamten Syntaxbaum durchlaufen befinden sich alle errechneten Constraints in einem ConstraintsSet.

5.3 Lösen der Constraints

Nach dem Abschluss der Berechnung der Constraints können aufgrund dieser Bedingungen die fehlenden Typen inferiert werden. Diesen Arbeitsschritt übernimmt der Unify-Algorithmus, dessen Spezifikation in Abbildung 1 dargestellt ist. Dieser wurde bereits in der Studienarbeit [Ott04] implementiert. Die Ausgabe des Unify-Algorithmus ist eine Menge von Pairs (siehe Kapitel 4). Im Erfolgsfall besteht diese Menge aus Paaren, welche jeweils einem TypPlaceholder einen konkreten Typen zuweisen.

TI$(\ Ass,$ Class$(\ \tau,$ extends$(\ \tau'\ ),$ *fdecls*, *mdecls*$)\ ) =$
 let
  $($Class$(\ \tau,$ extends$(\ \tau'\ ),$ *fdecls*$_t$, *mdecls*$_t\ ), ConS) =$
   TYPE$(Ass,$ Class$(\ \tau,$ extends$(\ \tau'\ ),$ *fdecls*, *mdecls*$)\ )$
  $(\{\ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n)\ \}, chk) =$ **UNIFY**$(\ ConS\ )$
 in
  $\{\ (cs_i, \sigma_i($ Class$(\ \tau,$ extends$(\ \tau'\ ),$ *fdecls*$_t$, *mdecls*$_t\ ))\ )\ 1 \leq i \leq n\ \}$

Abbildung 1: Typinferenzalgorithmus

5.4 Verarbeiten der Ergebnismenge

Nach einem erfolgreichen Durchlauf des Typinferenzalgorithmus muss im Anschluss die vorliegende Lösungsmenge in den Syntaxbaum übertragen werden. Dazu kann die in der Klasse "TypePlaceholder" enthaltene Funktionalität genutzt werden, welche die mit den Platzhaltern belegten Stellen im Syntaxbaum durch deren errechnete Typen ersetzt. Diese bietet allerdings den entscheidenden Nachteil, dass dieser Vorgang nicht umkehrbar ist. Ein Platzhalter kann somit nur einmalig mit einem Typ getauscht werden. Der Typinferenzalgorithmus ist allerdings so konzipiert, dass sein Ergebnis aus mehreren Lösungen bestehen kann. Das bedeutet, für einen Platzhalter kommen mehrere Typen in Frage. Mit der erwähnten Vorgehensweise könnte nun nur eine einzige Lösung übernommen werden. Zur Kontrolle des Algorithmus sollen jedoch sämtliche Lösungen ausgegeben werden. Dazu wurde eine neue Methode "printJavaCode" eingeführt. Sie durchläuft den Syntaxbaum und generiert dabei den zu der von diesem Baum repräsentierten Klasse zugehörigen Java-Quellcode. Entscheidend ist hierbei, dass der Methode eine der errechneten Lösungen übergeben wird. Die Methode gibt dann aufgrund dieser Informationen den entsprechenden Typ anstatt eines Platzhalters als Java-Quellcode aus. Diese Ausgabe lässt sich für beliebig viele Lösungen wiederholen.

6 Beispiel

In diesem Kapitel wird anhand eines Beispiels die Arbeitsweise des Typinferenzalgorithmus erläutert. Abbildung 2 zeigt den für dieses Beispiel verwendeten Java-Quellcode.

```
1 class Matrix extends Vector<Vector<Integer>>
2 {
3   operator = (m) -> {
4     return (f) -> {
5       return f.apply(this, m);
6     };
7   };
8 }
```

Abbildung 2: Java 8-Quellcode mit fehlenden Typangaben

Zuerst liest der Parser den Quellcode ein und erstellt einen Syntaxbaum. Auf diesem kann anschließend der Typinferenzalgorithmus arbeiten.
Mit den gegebenen Informationen kann der Typinferenzalgorithmus dann eine Menge von Lösungsmöglichkeiten ermitteln, welche den fehlenden Typangaben im Syntaxbaum Typen zuweisen. Diese Menge kann aber endlich viele Lösungen enthalten. Zur Visualisierung werden diese nacheinander in den Syntaxbaum eingesetzt, welcher daraufhin als Java-Quellcode ausgegeben wird. Für die in Abbildung 2 gezeigte Java-Klasse bestehen mehrere Möglichkeiten zur Typauflösung. Eine der vom Typinferenzalgorithmus ermittelten Lösungen ist in Abbildung 3 dargestellt.

```
1 class <M, L, K, E> Matrix extends Vector<Vector<java.lang.Integer>>
2 {
3   Fun1<Fun1<M, Fun2<L, Matrix, K>>, E> operator = (E m) -> {
4     return (Fun2<L, Matrix, K> f) -> {
5       return f.apply(this, m);;
6     };
7   };
8 }
```

Abbildung 3: Java-Quellcode mit rekonstruierten Typen

Den in Abbildung 2 gezeigten Quellcode kann der Typinferenzalgorithmus jedoch nicht eindeutig typisieren. Einige der untypisierten Variablen, wie zum Beispiel der Parameter “m“, werden nie in einem Kontext verwendet, der auf einen Typ schließen lässt. Diese Variablen erhalten daher einen generischen Typ.
Die Klasse “Matrix“ erbt in unserem Beispiel von “Vector<Vector<java.lang.Integer»“. Dadurch ergeben sich weitere Möglichkeiten der korrekten Typisierung des Beispiels. Eine davon ist in Abbildung 4 dargestellt.

```
1 class <M, L, K, E>Matrix extends Vector<Vector<java.lang.Integer>>
2 {
3 Fun1<Fun1<M, Fun2<L, Vector<Vector<java.lang.Integer>>, K>>, E> op = (E m)
      -> {
4 return (Fun2<L, Vector<Vector<java.lang.Integer>>, K> f) -> {
5 return f.apply(this, m);;
6 };
7 };
8 }
```

Abbildung 4: Java-Quellcode mit rekonstruierten Typen

7 Schlussbetrachtung

7.1 Fazit

Der in “Complete Typeinference for Java 8“ (siehe [Plüar]) definierte Typinferenzalgorithmus wurde im Zuge dieser Arbeit umgesetzt. Der Parser ist nun in der Lage Java 8 Quellcode einzulesen. Dazu musste auch der Syntaxbaum, welchen der Parser generiert, erweitert werden. Zusätzlich sind Testfälle und eine Ausgabe zur Anzeige der inferierten Typen implementiert worden.

7.2 Ausblick

Im Zuge der Arbeit konnte zwar der Typinferenzalgorithmus umgesetzt werden, der momentane Stand der Anwendung ist allerdings noch nicht für den produktiven Einsatz geeignet. Es ist noch keine Benutzerschnittstelle vorhanden und bisher nur eine für Testzwecke geeignete Ausgabe des inferierten Java-Quellcodes vorhanden. Der Typinferenzalgorithmus könnte zukünftig in einem Eclipse-Plugin zur Inferierung von innerhalb der Entwicklungsumgebung bearbeitetem Quellcode dienen. Dieses Plugin erspart dadurch dem Programmierer schreibarbeit indem es ausgelassene Typen automatisch rekonstruiert und in den Quellcode einsetzt. Eine weitere Möglichkeit wäre die direkte Erzeugung von Bytecode aus dem untypisierten Java. Dazu müsste das Typsystem um eine eingeschränkte Form von Durchschnittstypen erweitert werden [Plü08].

Literatur

[Bäu05] BÄUERLE, Jörg: *Typinferenz in Java*. 2005

[Her11] HERDEN, Olaf: *Vorlesung Compilerbau*. 04/2011

[Hol06] HOLZHERR, Timo: *Typinferenz in Java*. 2006

[Lüd07] LÜDTKE, Arne: *Java Typinferenz mit Wildcards*. 2007

[Naf13] NAFTALIN, Maurice: *Lambda FAQ*. `http://www.lambdafaq.org/`, 2013. – [Online; letzter Zugriff 30.01.2013]

[OP10] OPENJDK-PROJECT: *BWorld Robot Control Software*. `http://openjdk.java.net`, 2010. – [Online; letzter Zugriff 30.01.2013]

[Ott04] OTT, Thomas: *Typinferenz in Java*. 2004

[Plü08] PLÜMICKE, Martin: Intersection Types in Java. In: VEIGA, Luís (Hrsg.) ; AMARAL, Vasco (Hrsg.) ; HORSPOOL, Nigel (Hrsg.) ; CABRI, Giacomo (Hrsg.): *6th International Conference on Principles and Practices of Programming in Java* Bd. 347, 2008 (ACM International Conference Proceeding Series), S. 181–188

[Plü09] PLÜMICKE, Martin: Java type unification with wildcards. In: SEIPEL, Dietmar (Hrsg.) ; HANUS, Michael (Hrsg.) ; WOLF, Armin (Hrsg.): *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers* Bd. 5437, Springer-Verlag Heidelberg, 2009 (Lecture Notes in Artificial Intelligence), S. 223–240

[Plü10] PLÜMICKE, Martin: *Das JCC-Projket*. `http://www.ba-horb.de/ pl/JCC.html`, 2010. – [Online; letzter Zugriff 30.01.2013]

[Plüar] PLÜMICKE, Martin: *Complete type inference in Java*. to appear