Berufsakademie Stuttgart · Außenstelle Horb
Staatliche Studienakademie
University of Cooperative Education
Studiengang Informationstechnik

Studienarbeit

Typinferenz in Java

Student:

Timo Holzherr
Tübinger Str. 23
72108 Rottenburg

Studienjahrgang 2003

Ausbildungsbetrieb:

HYDMedia GmbH
Gartenstr. 87-89
72108 Rottenburg

Betreuer:

Prof. Dr. Martin Plümicke

Ehrenwörtliche Erklärung

(gemäß §19 (4) der Verordnung des Ministeriums für Wissenschaft und Kunst über die Ausbildung und Prüfung der Studierenden der Studienakademie im Land Baden-Württemberg vom 6. Februar 2001)

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Ort, Datum Unterschrift des Studierenden

Zusammenfassung

In der Programmiersprache *Java* ist der Programmierer gezwungen bei der Deklaration von Variablen einer Methode deren Typen explizit anzugeben. Meistens lassen sich diese Typen jedoch auch ohne eine explizite Definition aus dem Kontext berechnen.

Hierzu wurde in Vorgängerstudienarbeiten ein Prototyp eines Java-Compilers mit einem Typinferenzsystem entwickelt, der Typen eines Programms, die nicht explizit angegeben wurden, berechnet.

Die vorliegende Studienarbeit beschäftigt sich mit der Weiterentwicklung dieses Compilers und dessen Typinferenzalgorithmus.

Abstract

In the programming language *Java*, it is mandatory to define types of method-variables explicitly. In most cases however, these types can be calculated from the context without explicit type-definitions.

In order to make this possible, in preceeding student research projects students developed a prototype of a Java compiler containing a type inference system which is able to calculate types that aren't explicitly defined.

The present student research project deals with the enhancement of this compiler and its type inference algorithm.

Vorwort

Ganz besonders möchte ich mich bei meinem Betreuer Prof. Dr. Martin Plümicke für die Zeit, die er sich für Gepräche und Diskussionen über Implementierungsfragen genommen hat, bedanken. Die Studienarbeit bot eine interessante Mischung aus theoretischen Überlegungen und praktischer Implementierung komplexer Abläufe.

Des weiteren danke ich Jörg Bäuerle für die Einführung in das bestehende Projekt und die Betreuung dieser Studienarbeit.

Meinen Studienarbeits-Kommilitonen möchte ich für die gute Kooperation im Team danken. Nur so war es möglich sinnvolle Lösungen für projektübergreifende Probleme zu finden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Programmiersprache Java erwartet bei der Deklaration jeder Variablen eines Programms die explizite Angabe eines Typs. Diese Typangaben können vorallem bei komplexeren Ausdrücken für den Programmierer schnell lästig und zum Teil recht unübersichtlich und verwirrend werden. Mit der Einführung generischer Typen im JDK 1.5 von Sun wurden die anzugebenden Typterme noch komplexer. Hier wäre es für einen Java-Programmierer hilfreich, wenn er das Einfügen der Typinformationen dem Compiler selbst überlassen könnte.

1.2 Projektgeschichte

Der Java-Compiler, der die Basis der vorliegenden Studienarbeit bildet, wurde von Studenten des Kurses TIT2000 in einem Projekt der Vorlesung »Software-Engineering« entwickelt.

In den Vorgängerstudienarbeiten [Haa04] und [Rei03] wurde dieser Compiler um die Unterstützung generischer Datentypen erweitert.

Der in [Plu05] spezifizierte Unifikationsalgorithmus wurde im Rahmen der Studienarbeit [Ott04] implementiert.

Im Rahmen der Studienarbeiten [Bae05] und [Mel05] wurde der in [Plu05] spezifizierte Typrekonstruktionsalgorithmus (im Folgenden als TRA abegkürzt) in diesen Java-Compiler implementiert und um eine grafische Oberfläche für die Entwicklungsumgebung *Eclipse* erweitert.

1.3 Aufgabenbeschreibung

Aufgabe der vorliegenden Studienarbeit ist die Vervollständigung der bestehenden Implementierung des TRA zu einer β-Version. Zudem sollen der TRA und der Parser auf Interfaces, gebundene *(bounded)* generische Typen und *Packages* erweitert werden.

2 Theoretische Grundlagen

2.1 Compiler

Ein Compiler ist nach [Bae05] ein Programm, welches ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache umwandelt. Das zu compilierende Programm ist meist in einer höheren Programmiersprache geschrieben. Die Zielsprache ist häufig eine maschinennahe Sprache wie *Assembler*, *Bytecode* oder *Maschinensprache*.

Der Kompiliervorgang, der den Übersetzungsvorgang eines Programmes von einer Quellsprache in eine Zielsprache durchführt, kann in zwei Phasen unterteilt werden. Die Analyse- und die Synthesephase.

In der Analysephase wird das Quellprogramm zunächst mithilfe eines Parsers in eine hierarchische Struktur, den Syntaxbaum, gebracht. Danach folgt die semantische Analyse, welche diesen Syntaxbaum auf semantische Fehler untersucht. Nach diesem Vorgang ist bei erfolgreichem Abschluss die Analysephase beendet und ein Syntaxbaum entstanden, der ein fehlerfreies Quellprogramm maschinenunabhängig repräsentiert. Während der Analysephase kann in jeder Teilphase ein Fehler im Quellprogramm erkannt werden. Zur Verwaltung von im Quellprogramm definierten Bezeichnern und deren Zusatzinformationen verwenden Compiler Symboltabellen, in denen diese Informationen gesammelt werden können.

In der Synthesephase wird anhand des in der Analysephase entstandenen Syntaxbaumes der Programmcode der Zielsprache erzeugt. Diese Phase kann widerum in mehrere Teilphasen untergliedert sein.

2.2 Java

Die Programmiersprache *Java* [GJSB05] wurde ursprünglich für Geräte aus dem Unterhaltungsbereich entwickelt. Sie wird heutzutage jedoch in sehr vielen verschiedenen Bereichen, die von Webserveranwendungen bis hin zu Anwendungen auf mobilen Geräten wie Mobiltelefonen ablaufen.

Diese Programmiersprache ist eine Mischung aus interpretierter und compilierter Programmiersprache. Beim Kompiliervorgang wird aus einem Java-Programm Bytecode generiert, der plattform- und maschinenunabhängig ist. Ein Interpreter, die Java Virtual Machine (im folgenden auch JVM genannt), liest den generierten Bytecode zur Laufzeit ein und produziert für das System, auf dem die Applikation laufen soll, maschinenabhängigen Code. Aus diesem Grund sind Java-Programme auf allen Systemen lauffähig, für die es eine JVM gibt. Durch

den Mechanismus der Just-In-Time-Compilierung werden Laufzeiteinbußen, die interpretierte Sprachen normalerweise mit sich bringen, bei modernen JVMs kompensiert.

Die Syntax der Programmiersprache Java ist der Syntax der Programmiersprache *C* [KR90] sehr ähnlich. Java ist eine objektorientierte Programmiersprache, besitzt jedoch einige Eigenschaften nicht, die andere objektorientierte Programmiersprachen besitzen. Beispiele hierfür sind Zeiger und Mehrfachvererbung bei Klassen. Durch den objektorientierten Ansatz ist jedoch eine hohe Wiederverwendbarkeit und gute Strukturierung des entstandenen Codes gegeben.

2.3 Generische Datentypen

Für die Wiederverwendbarkeit objektorientierter Programme spielen generische Datentypen eine wichtige Rolle. Diese Datentypen ermöglichen es, Algorithmen so weit wie möglich von ihren Datentypen zu trennen. Algorithmen müssen dadurch nicht für neue Datentypen neu implementiert werden.

Generische Klassen sind Klassen, die generische Datentypen beinhalten. Bei der Instanzierung dieser Klassen können die generischen Datentypen spezifiziert werden. Die Klasse kann so je nach Anwendung parametrisiert werden.

Eine häufig verwendete Anwendung generischer Datentypen findet sich in Container-Klassen. Eine weitere gebräuchliche Anwendung besteht in Sortier-Algorithmen. Ein Sortier-Algorithmus kann dann unabhängig davon, welche Art von Datentypen er sortieren soll, diese Operation durchführen. Voraussetzung dafür ist jedoch, dass er die Information besitzt, wie die zu sortierenden Objekte zueinander in Relation stehen. Dies lässt sich in der Programmiersprache Java besonders elegant durch gebundene generische Datentypen realisieren.

```
class SortableData<E extends Comparable>{

        // Beinhaltet die Daten
        java.util.Vector<E> data=new java.util.Vector<E>();

        // Neues Element einfuegen
        void add(E element){
                data.addElement(element);
        }
        // Element aus der Menge laden
        E get(int index){
                return(data.elementAt(index));
        }

        // Daten sortieren
        void sort(){
                // Sortierung anhand der Informationen
```

```
                // der Methode E.compareTo(...)
        }
}
```

Das Interface *Comparable* der Java-API definiert die Methode *int compareTo(Object)*. Jedes Objekt, das über die Methode *add* in die Menge hinzugefügt werden kann, muss aufgrund des gebundenen Parameters das Interface *Comparable* implementieren. Dadurch steht den Objekten des Typs *E* die Methode *compareTo(Object)* zur Verfügung. So ist sicher gestellt, dass der Sortier-Algorithmus stets die Information besitzt, wie die zu sortierenden Objekte in Relation gesetzt werden können.

In Java können seit der Version 5.0 - auch Tiger-Java genannt - Klassen und Methoden durch generische Typen parametrisiert sein. Besonders bei statischen Methoden erhalten parametrisierte Methoden eine Schlüsselfunktion. Statische Methoden dürfen keine generischen Parameter von Klassen enthalten, da sie nicht auf Instanzen von Objekten ausgeführt werden.

Das folgende Beispiel zeigt eine prakische Anwendung generischer Parameter in statischen Methoden. Bei dem Beispiel soll es die Methode *getGreatestObject* ermöglichen, aus einer Menge von Zahlen (die in einem Vector gespeichert sind) das größte Element zu finden.

```
import java.util.Vector;

class Helper{
    static <N extends Number> N getGreatestObject(Vector<N> data){
        N greatest=null;
            for(N element:data){
                if(greatest!=null){
                    if(greatest.floatValue()<element.floatValue()){
                        greatest=element;
                    }
                }else{
                    greatest=element;
                }
            }
        return greatest;
    }
}
```

Der Rückgabetyp der Methode variiert je nach Aufruf. Wird der Methode beispielsweise ein *Vector<Integer>* übergeben, so wird automatisch ein Objekt vom Typ *Integer* zurückgegeben. Wird stattdessen ein *Vector<Float>* übergeben, ist der Return-Typ *Float*.

2.4 Typinferenz

Der Begriff *Inferenz* stammt nach [Bae05] vom englischen Begriff »inference« ab, was man als Schlussfolgerung oder Rückschluss übersetzen kann. Demnach bedeutet Typinferenz *Typrückschluss*.

Aus Sichtweise der Informatik bezeichnet Typinferenz das Folgern eines bestimmten Typs für einen gegebenen Ausdruck, dessen Typ nicht explizit angegeben wurde.

Typinferenzalgorithmen arbeiten meist direkt auf dem abstrakten Syntaxbaum eines Compilers und führen auf diesem die Typrekonstruktion durch. Aus diesem Grund sind sie Teil der semantischen Analyse von Compilern.

Der in den bestehenden Compiler eingebaute Typinferenzalgorithmus basiert auf das in [Plu05] definierte Typinferenzsystem. Der in der Studienarbeit [Bae05] implementierte Typinferenzalgorithmus rekonstruiert Typen mithilfe seiner Unteralgorithmen auf Basis des in [Ott04] implementierten Unifikationsalgorithmus TUnify.

Bei der Unifikation [BS01] werden Typpaare, die unifiziert werden sollen, anhand von Schlussregeln untersucht. Um Typen *gleich* zu machen, werden Substitutionen berechnet. Diese Substitutionen, auch Unifikatoren oder Unifier genannt, werden im TRA dazu verwendet, Typ-Platzhalter *(TypePlaceholders)* durch die konkreten berechneten Typen zu ersetzen.

Im folgenden Beispiel soll der Ablauf des TRA genauer erläutert werden. Zur Erklärung dieses Beispiels ist zu sagen, dass der vorliegende Compiler statt den gebräuchlichen primitiven Datentypen für numerische Operationen deren Wrapper-Klassen verwendet. (Genauere Erläuterungen unter → 3.5.8)

```
class Helper{
        public increment(number){
                return(number++);
        }
}
```

Bei der Deklaration der Methode *increment* wurde weder für den Rückgabewert noch für den Parameter *number* der Methode ein Typ angegeben. Aus diesem Grund erstellt der Parser beim Einlesen der Datei für jeden dieser Typen einen TypePlaceholder, der als Typ für die Variablen dient.

Anhand der Typrekonstruktionsregeln der Inkrement-Operation kann der TRA folgern, dass es sich bei der Variablen *number* um eine Variable vom Typ Integer oder einer Subklasse des Typs Integer handeln muss. Der Unifikationsalgorithmus berechnet mit dieser Information die möglichen Unifikatoren. In diesem Fall wird für den TypePlaceholder (es wird angenommen, dass der TypePlaceholder in diesem Beispiel die Bezeichnung *A* trägt) des Methodenparamters lediglich ein Unifikator, $\{(A \rightarrow Integer)\}$, berechnet.

Bei der Rückgabe des durch die Inkrement-Operation berechneten Wertes wird der Rückgabetyp der Methode mit dem Rückgabetyp der Inkrement-Operation unifiziert. (Hier wird angenommen, dass der TypePlaceholder des Rückgabetyps den Bezeichner *B* trägt.) Die in diesem Fall greifenden Typrekonstruktions-Regeln legen fest, dass es sich bei dem Rückgabetyp der Methode um den Rückgabetyp der Inkrement-Operation oder eine Superklasse dieses Typs handeln muss. Mit dieser Information berechnet der Unifikationsalgorithmus die möglichen Substitutionen. Die möglichen Unifikatoren sind hier $\{(B \rightarrow Integer), (B \rightarrow$

$Number), (B \rightarrow Object)\}$. (Für genauere Informationen zur Vererbungshierarchie der Klasse *java.lang.Integer* wird auf → 3.5.4 verwiesen).

3 Weiterentwicklungen des Typrekonstruktionsalgorithmus

3.1 Design-Änderungen

3.1.1 Debug-Ausgaben

Für die Debug-Ausgaben des Compilers wurde der Log-Mechanismus *log4j* eingeführt. Der bisher eingesetzte Mechanismus bestand darin, Log-Level beim Aufruf der Applikation zu übergeben. Es war nicht möglich die Debug-Ausgaben der einzelnen Teilschritte wie Parsen, Typinferenz und Codegenerierung separat zu steuern. An einigen Stellen wurde das Implementieren der Log-Abfrage vergessen, sodass das Einstellen des Log-Levels keine Auswirkung darauf hatte.

Durch den Mechanismus von *log4j* ist es nun möglich in der Konfigurations-Datei *log4j.xml* für jeden Teilschritt des Compilers vom *Parser* bis hin zur *Bytecodegenerierung* separate Log-Levels einzustellen.

3.1.2 TyploseVariable vs. TypePlaceholder

Die Klasse *TyploseVariable* wurde in *TypePlaceholder* umbenannt. Die Bezeichnung *TyploseVariable* ($\rightarrow$ siehe [Bae05]) ist irreführend, da es sich bei einer Variablen vom Typ *TyploseVariable* keineswegs um eine Variable handelt, die keinen Typ besitzt, sondern um eine Variable, deren Typ noch nicht vom TRA berechnet wurde.

3.2 Interfaces

3.2.1 Einführung

Eine der grundlegenden Aufgaben dieser Studienarbeit bestand in der Erweiterung des Compilers um das Java-Konstrukt *Interface*.
Bisher war dieses Konstrukt dem Typinferenz-Compiler und somit seinen Untereinheiten Parser und TRA nicht bekannt. Um eine Unterstützung dieser Funktion im TRA zu ermöglichen, musste zuerst der Parser auf Interfaces und deren Eigenschaften wie Superinterfaces, Methoden, Konstanten und generische Typen erweitert werden.

Dazu wurde eine Klasse Interface angelegt, die die nötigen Eigenschaften eines Interfaces abbilden kann. Um eine gemeinsame Basis für Klassen und Interfaces zu schaffen erben sowohl *Class* als auch *Interface* von der Klasse *AClassOrInterface*, die die gemeinsamen Eigenschaften beider Konstrukte beinhaltet.

Da Interfaces keine ausprogrammierten Methoden sondern nur Methodendeklarationen beinhalten dürfen, ist es nicht möglich, eine Typrekonstruktion auf Interfaces durchzuführen. So war es nicht nötig, den TRA auf Interfaces anzuwenden. Vielmehr war es die Aufgabe, die Typrekonstruktion einer Klasse durchzuführen, die Interfaces verwendet.

3.2.2 Finite Closure

Anhand des folgenden Beispiels soll eine Anwendung des TRA mit Klassen und Interfaces erläutert werden.

```
public interface Intf1{
}
public interface Intf2{
}
public class Class1 implements Intf1, Intf2{
        void doSomething(){ ... }
}

class Test{
        public test(x){
                x.doSomething();
                return(x);
        }
}
```

Im vorliegenden Programm sind zwei Interfaces und zwei Klassen definiert. Die Interfaces und die Klasse *Class1* dienen zur Bildung einer Vererbungsstruktur, die in der Klasse *Test* verwendet werden soll. Die Klasse *Class1* besitzt eine Methode *doSomething*, anhand deren der TRA den Typ des Aufrufparameters und den Rückgabetyp der Methode *test* der Klasse *Test* erkennen kann.

Die korrekte Ergebnismenge der Typrekonstruktion der Methode *Test.test(x)*, sieht folgendermaßen aus:

- test: Class1 –> Class1
- test: Class1 –> java.lang.Object
- test: Class1 –> Intf1
- test: Class1 –> Intf2

Bei dem Methodenparameter *x* kann es sich, wie oben zu erkennen ist, nur um den Typ *Class1* handeln. Nur diese Klasse besitzt die Methode *doSomething*. Als Rückgabetyp sind jedoch alle vier Typen *Class1*, *java.lang.Object*, *Intf1*, *Intf2* möglich, da diese Superklassen bzw. Superinterfaces der Klasse *Class1* sind.

Damit der TRA die Vererbungsbeziehungen zwischen den Klassen und Interfaces des zu compilierenden Programms kennt und das oben geschilderte Problem lösen kann, mussten die Interfaces in die Menge der *Finite Closure* [Ott04] aufgenommen werden.

Implementiert eine Klasse - im vorangegangenen Beispiel die Klasse *Class1* - eines oder mehrere Interfaces, so muss je Superinterface ein Paar *Pair=(Klasse,Superinterface)* in der Menge der *Finite Closure* abgelegt werden.

Implementiert ein Interface eines oder mehrere Interfaces, muss auch für dieses je Superinterface ein Paar *Pair=(Interface,Superinterface)* in der Menge der *Finite Closure* abgelegt werden.

Dadurch wurde eine korrekte Abbildung der Vererbungshierarchie der Interfaces ermöglicht.

3.2.3 Typannahmen

Durch die Erweiterung der Finite Closure wurde erreicht, dass Interfaces bei der Verwendung ihrer implementierenden Klassen auch als mögliche Supertypen erkannt werden. Wird jedoch ein Interface direkt verwendet indem beispielsweise ein Aufruf einer durch ein Interface definierte Methode erfolgt, wird diese Methode bisher nicht gefunden.

Folgendes Beispiel soll anhand eines *MethodCalls* verdeutlichen, welche Veränderungen bei der Typrekonstruktion von Interfaces nötig waren.

```
interface Intf{
        doSomething();
}
class Test{
        public test(x){
                x.doSomething();
        }
}
```

Die erste Methode, die typinferiert wird, ist die Methode *test*. Dabei wird durch alle Anweisungen der Methode iteriert. Auf diese Anweisungen werden jeweils die Unteralgorithmen der Typrekonstruktion angewandt. Im gegebenen Fall wird in der Menge der *MethodIntersectionTypes* in allen dort definierten Klassen nach der Methode *doSomething(x)* gesucht.

Die Menge der *MethodIntersectionTypes* wird im Algorithmus *TRProg* [Plu05] mit den Typannahmen der jeweils aktuellen Klasse befüllt. Da jedoch die Interfaces selbst nicht typinferiert werden, wird der Algorithmus *TRProg* nicht darauf angewandt. Somit befinden sich die Interfaces nicht in der Menge der *MethodIntersectionTypes*.

Damit die Methoden- und Konstantendeklarationen trotzdem vom TRA gefunden werden, müssen diese vor dem Ablauf des TRA bereits in die Menge der *MethodIntersectionTypes* geladen werden. Dazu wurde der Klasse *Interface* die Methode *addThisToAssumptions* hinzugefügt, die den bisher erstellten BasicAssumptions die Methoden, Konstanten und generischen

Variablen des Interfaces in Form von Typannahmen hinzufügt.

3.3 Generische Methoden- und Klassenparameter

Eine weitere grundlegende Aufgabe dieser Studienarbeit war es, den Typinferenz-Compiler um generische Methoden- und Klassenparameter zu erweitern.

3.3.1 Generische Methodenparameter

Erweiterung des Parsers

Damit der Compiler die Syntax von generischen Methodenparametern akzeptiert, musste der Parser um dieses Konstrukt erweitert werden. Der Klasse *Method* wurde ein Attribut *genericMethodParameters* vom Typ *Vector<GenericTypeVar>* hinzugefügt. Dieses Attribut beinhaltet nach dem Parsen alle im zu compilierenden Programm definierten generischen Parameter, die die eingelesene Methode besitzt.

Um *Bounded generic Typevars* (siehe → 2.3) in der Parameterliste von Methoden im abstrakten Syntaxbaum darstellen zu können wurde die Klasse *BoundedGenericTypeVar*, die auf die Klasse *GenericTypeVar* basiert, erstellt. Sie besitzt einen Vector *bounds*, welcher alle *Bounds*, die im zu compilierenden Programm angegeben wurde beinhalten kann.

Die Klassenhierarchie des Typs *BoundedGenericTypeVar* sieht nun folgendermaßen aus:

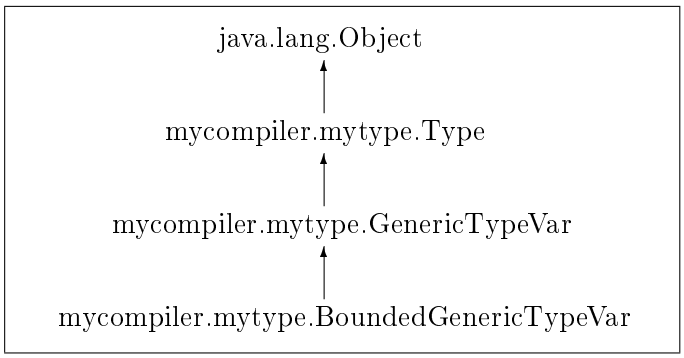

Abbildung 3.1: Vererbungshierarchie der Klasse *BoundedGenericTypeVar*

So ist es möglich, im Attribut *genericMethodParameters* der Klasse *Method*, welches die generischen Parameter der Methode beinhaltet, auch *BoundedGenericTypeVars* abzulegen.

Erweiterungen des TRA

Damit die Erweiterungen der Datenstrukturen Auswirkungen auf die Typrekonstruktion bekommen, musste analysiert werden, an welchen Stellen der Typrekonstuktion eine Anpassung des TRA erforderlich war.

Die in → 4.2 erwähnte Korrektur des TRA prüft bei jeder Deklaration einer Variablen in einer Methode, ob es sich wirklich um einen *RefType* oder um eine falsch erkannte *GenericTypeVar* handelt. Da falls wirklich eine Fehlerkennung stattgefunden hat, die komplette Typvariable durch eine Referenz auf die echte *GenericTypeVar* ersetzt wird, werden auch die *Bounds* der Variablen mit in die Deklaration übernommen. Hier musste demnach keine Korrektur erfolgen.

Wird jedoch eine Methode aufgerufen, die generische Methodeparameter besitzt, so müssen diese mit in die Behandlung des Methodenaufrufs einbezogen werden.

Dabei müssen alle generischen Parameter der Methode durch einen neu erstellten *TypePlaceholder* ersetzt werden, damit dieser sich wie ein generischer Typ verhält.

Die darauf folgende Unifikation der Methode ersetzt den *TypePlaceholder* durch den passenden Typ.

So wird im folgenden Beispiel der Variable *a* erfolgreich ein vom Typ *String* zugewiesen.

```
class GenericMethodClass {
        <E> id(E elem){
                return(elem);
        }
}
class Tester {
        test(GenericMethodClass x){
                a;
                a=x.id("String");
        }
}
```

Gebundene (bounded) generische Methodenparameter im TRA

Handelt es sich bei dem generischen Parameter der Methode um einen gebundenen *(bounded)* Parameter, müssen dessen Bounds als Zusatzbedingung mit in die Unifikation eingehen.
Am folgenden Beispiel sollen die Unifikationsbedingungen erläutert werden.

```
class NumberManager {
        ...
        <E extends Number> setElement(E elem){
                return(elem);
        }
}
class Application {
        test(Integer i){
                x;
                x.setElement(i);
        }
}
```

Für die Untersuchung des Beispiels wird angenommen, dass die Variable *x* den *TypePlaceholder* **A** bekommen hat.

Die erste Bedingung der Unifikation bildet sich dann aus dem Paar *(A, NumberManager)*. Die zweite Bedingung der Unifikation bildet sich aus dem Parametertyp, der im Methodenaufruf eingesetzt ist (Integer) und dem Typ, des ersten Methodenparameters(E). Da es sich bei dem Methodenparameter hier um eine generische Variable handelt, wird wie in → 3.3.1 beschrieben die generische Variable durch einen neuen *TypePlaceholder* ersetzt. Für diesen Fall wird definiert, dass dieser *TypePlaceholder* **B** lautet. Demnach geht als zweite Bedingung der Unifikation das Paar *(Integer, B)* ein.

Hätte der generische Methodenparameter keine *Bounds*, wären die Unifikationsbedingungen komplett. Da dies jedoch nicht der Fall ist, müssen für alle *Bounds* zusätzliche Unifikations-Paare erzeugt werden. In diesem Fall gibt es nur ein *Bound*, *Number*. Die zusätzliche Unifikationsbedingung lautet deshalb *(B, Number)*.

Mit den hierdurch gewonnenen Unifikationsbedingungen

`{(A, NumberManager), (Integer, B), (B, Number)}`

wird die Unifikation gestartet. Als Ergebnis liefert der Algorithmus den Unifikator {*(A → NumberManager), (B → Integer)*}. Der Methodenaufruf wurde somit korrekt aufgelöst und typinferiert.

3.3.2 Automatische Erzeugung von generischen Methodenparametern

Es gibt Methoden, die der TRA nicht komplett inferieren kann. Dieser Fall tritt beispielsweise auf, wenn eine Variable außer bei der (typlosen) Deklaration nicht verwendet wird. Im Ergebnis des TRA bleibt die Variable vom Typ *TypePlaceholder*.

Da *TypePlaceholders* jedoch keine rellen Typen sondern lediglich Hilfskonstrukte der Typinferenz sind, können diese im Bytecode nicht abgebildet werden. Wegen dieser Eigenschaft sind Programme, die nicht komplett typinferiert wurden, nicht im Bytecode abbildbar. Selbst wenn lediglich *ein* Parameter einer Methode nicht inferiert werden konnte, ist keine Compilierung möglich.

Am Beispiel der Identitätsfunktion soll das Problem und verschiedene Lösungsansätze diskutiert werden. Die *Identitätsfunktion* ist »eine identische Abbildung oder die Identität einer Funktion, die genau ihr Argument zurückgibt [...]« [Wik06]

```
class Identity<X>{
        id_string(para){
                return(para);
        }
        id_integer(para){
                return(para);
        }
}
```

Im gezeigten Beispiel besitzt die Klasse *Identity* zwei Methoden, *id_string* und *id_integer*. Beide bilden die *Identitätsfunktion* ab. Um zu prüfen, ob eine Lösung für dieses Problem hinreichend akzeptabel ist, wird das Ergebnis dieser Lösung mit der folgenden expliziten Typisierung verglichen, die hier als *Intention des Anwenders* gegeben wird.

```
class Identity<X>{
        String id_string(String para){
                return(para);
        }
        Integer id_integer(Integer para){
                return(para);
        }
}
```

Es muss also in der gefundenen Lösung möglich sein, der Methode *id_string* einen *String* zu übergeben und einen Typ $\leq$ *String* zu bekommen. Der Rückgabetyp kann also entweder die Klasse *String* oder eine Subklasse der Klasse *String* sein. Ebenso ist es eine Anforderung an die Typen der Methode *id_integer*, ein *Integer*-Objekt zu übergeben und einen Typ $\leq$ *Integer* zurück zu bekommen.

Im Folgenden werden die verschiedenen Lösungsansätze vorgestellt.

Ansatz I: Substitution durch den Typ *Object*

Der erste Ansatz besteht darin, alle nicht inferierbaren Typen durch den Typ *Object* zu ersetzen. Da *Object* der generellste Typ im Typsystem von Java ist, können als Parameter des Methodenaufrufs alle möglichen Typen eingesetzt werden. Da keine Typinferenz durchführbar war, gibt es auf den Typ der Variable keine Einschränkung. Somit kann das Programm erfolgreich compiliert werden.

Als Rückgabetyp der Methode ergäbe sich demnach auch der Typ *Object*. Wird die oben dargestellte Klasse jedoch instanziert, die Methode *id_string* oder *id_integer* aufgerufen, so kann nur der Typ *Object* zurückgegeben werden.

Folgende Anwendung der Klasse, die im Falle der expliziten Variablendeklaration möglich gewesen wäre, ist in diesem Ansatz nicht möglich.

```
class Instanzer{
        String test(){
                x;
                return(x.id_string("foo"));
        }
}
```

Es wäre ein explizites Typecast erforderlich, welches durch die Typinferenz jedoch nicht berechnet werden kann.

Aus diesem Grund stellt dieser Ansatz keine Lösung für das Problem dar.

Ansatz II: Substitution durch die generische Klassenvariable

Dieser Ansatz besteht darin, nichterkannte Typen durch die generische Klassenvariable zu ersetzen. Bei der Instanzierung und Verwendung einer Methode würde das Problem korrekt gelöst werden. Der TRA würde die Typen wie im folgenden Codebeispiel dargestellt erkennen.

```
class Instanzer{
        String test(){
                Identity<String> x;
                return(x.id_string("foo"));
        }
}
```

Der Rückgabewert würde mit dem erforderlichen Rückgabetyp *String* übereinstimmen und das Problem wäre für diesen Fall gelöst.

Werden jedoch beide Methoden nacheinander auf das Objekt *x* ausgeführt, zeigt sich, dass auch dieser Ansatz keine Lösung des Problems darstellt.

```
class Instanzer{
        String test(){
                x;                                  // Zeile 1
                String s  = x.id_string("foo");     // Zeile 2
                Integer i = x.id_integer(1);        // Zeile 3
        }
}
```

In *Zeile 2* wird für den Typ von *x* durch den TRA die Klasse *Identity<String>* berechnet. Der Methodenaufruf in *Zeile 3* kann deshalb nicht ausgeführt werden.

Aus diesem Grund stellt auch dieser Ansatz keine gute Lösung für das Problem dar.

Ansatz III: Substitution durch einen neuen generischer Methodeparameter

Im dritte Ansatz generiert der TRA für jeden nicht inferierbaren *TypePlaceholder* einen neuen generischen Methodenparameter, durch den nach erfolgter Typinferierung der übrig gebliebene *TypePlaceholder* ersetzt wird. Der generische Methodenparameter muss nach der Substitution in die Menge der generischen Parameter der Methode eingetragen werden. Nach der erfolgten Typinferenz und Substitution würde die Typinformation der Klasse *Identity* folgendermaßen aussehen:

```
class Identity<X>{
        <E> E id_string(E para){
                return(para);
        }
        <E> E id_integer(E para){
                return(para);
        }
}
```

Erfolgt dann wie im zweiten Ansatz dargestellt, ein Aufruf beider Methoden auf dasselbe Objekt, werden aufgrund der generischen Methodenparameter die jeweiligen Typen für die generische Variable **E** eingesetzt. Beim Aufruf der Methode *id_string* mit einem Parameter vom Typ *String* gibt die Methode ein Objekt vom Typ *String* zurück. Beim Aufruf der Methode *id_integer* mit einem Parameter vom Typ *Integer* gibt diese Methode ein Objekt vom Typ *Integer* zurück.

Dieser Ansatz genügt den Anforderungen der expliziten Typdefinition und wurde deshalb realisiert.

Realisierung

Im Algorithmus *TRStart* [Plu05] wird für jede Methode der aktuellen Klasse durch den Aufruf von *TRNextMeth* [Plu05] das Ergebnis der Typinferenz berechnet. Dieses besteht aus einer Menge von Typannahmen. Um den zuvor erwähnten dritten Ansatz zu realisieren, muss diese Ergebnismenge auf *TypePlaceholders* untersucht werden. Besitzt die im aktuellen Durchlauf von *TRNextMeth* inferierte Methode nach der Substitution der erkannten *TypePlaceholders* immernoch nichterkannte *TypePlaceholders*, so wird jeder *TypePlaceholder* durch einen neuen generischen Typ ersetzt. Dieser generische Typ wird zu der Menge der Methodenparameter hinzugefügt.

Dadurch sind für diesen Parameter uneingeschränkt alle existierenden Typen möglich. Bei der wie im Beispiel der *Identität* dargestellten Rückgabe der Variablen ist deren Typ identisch mit dem Eingabeparameter der Methode.

End-Analyse der gewählten Lösung

Die Wahl des dritten Ansatzes bringt Folgen für die korrekte Inferierbarkeit von Programmen mit sich. Da bei der Inferierung der Typen wie in → 3.3.2 beschrieben nicht angenommen werden darf, dass es sich bei dem Typ eines nichterkannten Type-Placeholders um einen generischen Klassenparameter handelt, werden nicht erkannte Typen stets durch generische Methodenparameter ersetzt.

War die Intention des Entwicklers jedoch, für diese Variable den generischen Typ einer Klasse zu verwenden, macht der TRA in diesem Fall eine Fehlentscheidung. Obwohl bei einer Instanzierung für den Klassenparameter ein expliziter Typ angegeben werden muss, ist wegen des generischen Methodenparametes im Programm der Aufruf der Methode mit beliebigen Typen erlaubt. Da jedoch keine Restriktion im Programmcode die Voraussetzung geschaffen hat, dass es sich bei dem Typ der nichterkannten Variable um einen generischen Klassenparameter handeln muss, kann man davon ausgehen, dass solch eine Fehlentscheidung des TRA in diesem Fall vertretbar ist.

Eine ähnliche Situation liegt vor, wenn die zu inferierende Methode bereits einen generischen Methodenparameter besitzt. Der TRA darf nicht automatisch Schlussfolgern, dass es sich bei einem nichterkannten TypePlaceholder um einen bereits vorhandenen und im Quellcode des

Programmes definierten generischen Methodenparameter handelt. Dies hat zur Folge, dass der TRA selbst wenn der Programmierer die Intention hat, den bereits deklarierten generischen Typ zu verwenden, einen neuen generischen Methodenparameter in die Methodendeklaration einfügt.

Diese Fehlentscheidung hat ebenso (wie die Fehlentscheidung bei generischen Klassenparametern) geringe Auswirkungen auf die Verwendbarkeit des entstandenen Programms.

Ausblick

Eine weitere Anwendung bietet sich in der Kapselung nicht-unifizierbarer Typen in generischen Methodenparametern. Das folgende Beispiel soll diesen Anwendungsfall verdeutlichen.

```
import java.util.Vector;

public class UseCase{
        public void method(para){
                Object anything;                    // Zeile 1
                para.addElement("String");          // Zeile 2
                para.compareTo(anything);           // Zeile 3
        }
}
```

Die zweite Zeile des Beispiels setzt voraus, dass es sich bei dem Parameter *para* um den Typ Vector<String> oder eine Subklasse dieses Typs handelt.

In der Menge der Typannahmen ist also notiert, dass es sich bei *para* um eine Klasse eines solchen Typs handeln muss. Wird nun die Typrekonstruktion auf die dritte Zeile angewandt, wird bisher bei der Unifikation ein Fehler entstehen. Es gibt keine Klasse, die die Eigenschaft besitzt, sowohl von der Klasse *String* als auch vom Interface *Comparable*, welches die Methode *compareTo(Object)* definiert, zu erben. Um trotzdem dem Anwender eine Lösung zu bieten, die keinen Fehler verursacht, besteht die Möglichkeit, einen neuen generischen Parameter einzuführen, welcher folgende Spezifikation besitzt:

```
<X extends Vector<String> & Comparable>
```

Um dies zu ermöglichen, ist es nötig, eine Subklasse der Klasse TypePlaceholder zu erzeugen. Diese muss die Menge der Superklassen beinhalten, die die gesuchte Klasse implementieren muss. Wird im weiteren Verlauf der Typrekonstruktion keine Klasse gefunden, die diesen Eigenschaften genügt, kann der TypePlaceholder (der jetzt bounds besitzt) durch eine *BoundedGenericTypeVar* ersetzt werden. Die Menge der Bounds der *BoundedGenericTypeVar* entspricht dann den Bounds des TypePlaceholders.

Dieser Vorgang darf jedoch nur ausgeführt werden, wenn sich in der Menge der Bounds, die entstehen würde, maximal eine Klasse befindet. Alle anderen Bounds müssen wegen der fehlenden Mehrfachvererbung in Java Interfaces sein.

Da eine solche Unterscheidung im Rahmen dieser Studienarbeit zu komplex geworden wäre, wurde auf die Implementierung dieser Funktion verzichtet.

Die Implementierung dieser Funktion kann die Grundlage einer weiterführenden Studienarbeit sein.

3.3.3 Gebundene *(bounded)* generische Klassenparameter

Klassen können in Java 5.0 auch gebundene generische Klassenparameter besitzen. Dabei wird wie bei den generischen Methodenparameter der Typ einer generischen Variable durch *Bounds* eingeschränkt.

```
class GenericClass<X extends Integer> {
        ...
}
```

Um solche Deklarationen zu ermöglichen, musste der Parser auf gebundene Klassenparameter erweitert werden. Hierzu konnte die Teilgrammatik verwendet werden, die bei generischen gebundenen Methodenparametern die Syntax definiert.

Dadurch lässt der Parser *bounds* in der Deklaration von Klassen zu. Damit diese beim Ablauf des TRA berücksichtigt werden, musste die Abhandlung eines Methodenaufrufs geändert werden. Dort musste - wie in → 3.3.1 beschrieben ähnlich dem Aufruf von Methoden mit gebundenen Methodenparametern - jede Bound mit als Zusatzbedingung in die Unifikation eingehen.

Bei der Verwendung des generischen Typs bei Deklarationen von Methodenparametern und Variablen muss keine zusätzliche Unterscheidung zwischen generischen und gebundenen generischen Typen erfolgen, da die Unifikation selbst die *bounds* berücksichtigt.

3.4 Packages

In der Programmiersprache Java kann man Klassen durch ihren einfachen Klassennamen oder durch den *Fully-Qualified Name* referenzieren. Ein Klassenname ist dann Fully-Qualified, wenn er den Paketnamen der Klasse beinhaltet.

Die vorliegende Implementierung des TRA arbeitet bisher nur mit einfachen Klassen- und Typnamen. Eine Verwendung von *Fully-Qualified Names* war bisher nicht möglich. Dadurch sind die Paketinformation referenzierter Klassen an keiner Stelle des Compilers bekannt.

Diese Information ist jedoch für die Generierung des Bytecodes von Bedeutung. Wird bei der Generierung des Bytecodes nur die simple Form des Klassennamens abgelegt, findet die JRE die Klasse zur Laufzeit nicht.
Eine Deklaration eines Vectors kann im Quellcode folgendermaßen aussehen:

```
Vector x=new Vector();
```

Damit diese Information jedoch korrekt in den Bytecode geschrieben werden kann, muss intern während der Bytecodegenerierung folgende Anweisung in den Datenstrukturen vorhanden sein.

```
java.util.Vector x=new java.util.Vector();
```

Der Compiler muss also an jeder Stelle des Quellcodes simple Klassennamen zu fully-qualified Klassennamen machen.

Diese Umwandlung wurde in den Ablauf des Compilers zwischen den Parse-Vorgang und die Typrekonstruktion eingebaut. Für die Umwandlung konnte die Liste der zu Importierenden Klassen verwendet werden. Dort sind alle Klassen, die das Programm aus der API von Java referenzieren kann, in ihrer *Fully-Qualified*-Form enthalten.

3.5 BasicAssumptions und Imports

3.5.1 Imports

In Java ist es wie in anderen objektorientierten Programmiersprachen möglich, Klassen der API der Programmiersprache zu referenzieren. Diese Klassen wurden im bestehenden Compiler über eine Menge an Typannahmen abgebildet, die vor dem Ablauf des TRA die Menge der Basis-Typannahmen - BasicAssumptions genannt - bildet.

3.5.2 Problemstellung

Diese Menge wurde bisher durch im Quellcode des Compilers fest codierte Anweisungen erstellt. Für jede Klasse wurde mit zehn bis zwanzig Anweisungen je Basistyp die Menge der Typannahmen erstellt. Es konnten somit nur Klassen referenziert werden, die zuvor im Quellcode des Compilers definiert wurden.

Diese Art der Abbildung der Java-API ist zwar für einen ersten Prototypen ausreichend, für die zukünftige Wartung und Weiterentwicklung jedoch zu umständlich und unflexibel. Um neue Klassen in die API einzupflegen war es bisher nötig, den Quellcode explizit zu modifizieren. Da der Compiler jedoch bei Demonstrationen mit dem Eclipse-Plugin aus einem JAR-Archiv geladen wird, bestand keine Möglichkeit, während der Demonstration kurzfristig neue Basistypen zu erstellen. Der Umfang an Anweisungen, der benötigt wurde, um eine Basisklasse zu definieren, war zudem zu komplex um dies kurzfristig durchzuführen.

3.5.3 Lösung

Um dieses Problem zu beheben, wurde im Parser das Schlüsselwort *import* implementiert. Beinhaltet das zu compilierende Programm eine Import-Anweisung, wird die dort angegebene Klasse mithilfe der Java-Runtime-Library nun direkt in Echtzeit, inklusive aller deklarierten Methoden, Attributen und Eigenschaften wie Superklassen, Superinterfaces und generischen Variablen in die Menge der BasicAssumptions geladen. Mit der Klasse *Class* bietet die Laufzeitumgebung von Java eine umfangreiche Schnittstelle, die dies ermöglicht.

Damit Basisklassen jedoch nicht dauerhaft in der Menge der vom Parser berechneten Klassen enthalten sind, müssen alle Basisklassen vor der Ausführung der Algorithmen des TRA wieder aus der Menge der Klassen entfernt werden. Ansonsten würde der TRA versuchen, auch bei allen Basisklassen eine Typrekonstruktion durchführen.

Bisher wurden diese Klassen jedoch durch feste Anweisungen im Quellcode und somit ohne Automatismus aus der Menge der Klassen entfernt. Wurde ein weiterer BasisTyp hinzugefügt, musste in der Subroutine, die die Typannahmen wieder entfernt, diese Klasse ebenfalls explizit definiert werden.

Da jedoch nun durch die *import*-Anweisung beliebige Klassen in die Menge der Basis-Typen geladen werden können, musste hierfür eine alternative Strategie entwickelt werden.

Werden compilierte Java-Klassen oder Basis-Klassen im abstrakten Syntaxbaum und in den Datenstrukturen abgelegt, werden die Klasseninformationen in einer Instanz der Klasse *mycompiler.myclass.Class* abgelegt.

Die eleganteste Lösung zur Unterscheidung von Klassen, die vom Parser gelesen wurden, und importierten Basisklassen, zeigte sich in der Verwendung verschiedener Typen für die beiden Klassenarten.

Die vom Parser eingelesenen Klassen werden nach wie vor in Objekten des Typs *Class* abgelegt. Klassen die der Compiler durch Import-Anweisungen bekommt, werden in Objekten des neuen Typs *BasicAssumptionClass* abgelegt. Da die Klasse BasicAssumptionClass eine Subklasse der Klasse *Class* ist, können die *BasicAssumptionClass*-Objekte im Ablauf der Algorithmen wie *Class*-Objekte behandelt werden.

Beim Entfernen der Basisklassen kann nun bei allen Klassen des Klassenvektors mit dem **instanceof**-Operator geprüft werden, ob es sich um eine *BasicAssumptionClass* oder eine vom Parser eingelesene Klasse vom Typ *Class* handelt. Die Objekte vom Typ *BasicAssumptionClass* können so erkannt und entfernt werden.

3.5.4 Kaskadierung von Imports

Durch die Typhierarchie in objektorientierten Programmiersprachen kann eine Import-Anweisung die Abarbeitung weiterer Import-Anweisung erfordern. Am Beispiel der Klasse *Integer* ist dies gut nachvollziehbar:

Bei der Abarbeitung von Import-Deklarationen werden Klasseneigenschaften wie Methoden, Attribute, Generische Parametertypen, Superklassen und Superinterfaces ausgelesen und gespeichert. Bei der Abarbeitung der Klasse *java.lang.Integer* wird die Superklasse *java.lang.Number* gefunden. Da der TRA die Klasse *java.lang.Number* jedoch nicht in der Liste der Import-Deklarationen hat und sie somit später auch nicht verarbeiten könnte, muss die Klasse *java.lang.Number* selbst wieder wie eine explizit definierte Import-Deklaration abgearbeitet werden. Die Klasse *java.lang.Number* besitzt selbst eine vorerst unbekannte Superklasse *java.lang.Object*. Auch diese muss wieder wie eine in der Import-Liste notierte Klasse behandelt

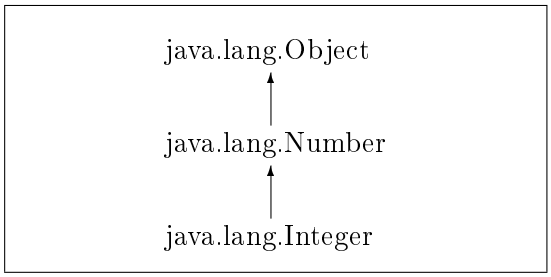

Abbildung 3.2: Vererbungshierarchie der Klasse *Integer*

und in die Datenstruktur geladen werden. Erst wenn die aktuell verarbeitete Klasse keine Superklasse besitzt - diese Vorraussetzung erfüllt nur die Klasse *java.lang.Object* - oder wenn die Superklasse der aktuellen Klasse bereits importiert wurde ist die Verarbeitung beendet.

3.5.5 *Superklasse* Object

In der Programmiersprache Java erben *alle* Klassen direkt oder indirekt von der Klasse *java.lang.Object*. Diese Eigenschaft spielt prinzipiell für die Ergebnismenge des TRA eine Rolle. Bisher wurde diese Eigenschaft im TRA nicht berücksichtigt. Durch die Implementierung der *Import*-Anweisung ist die Klasse *java.lang.Object* stets die indirekte oder direkte Superklasse aller importierten Klassen.

Da jedoch bisher eine Klasse, wenn keine Superklasse angegeben wurde, nicht wie gewünscht die Superklasse *java.lang.Object*, sondern keine Superklasse besaß, musste hier eine Veränderung im Konstruktor der Klasse *mycompiler.myclass.Class* vorgenommen werden.

Nun besitzen alle Klassen, bei denen keine Superklasse während dem Erstellen angegeben wurde, automatisch die Superklasse *java.lang.Object*.

3.5.6 Wildcards

Die in Java bestehende Möglichkeit, *Wildcards* bei einer *Import*-Deklaration zu verwenden (siehe Beispiel), wurde hier aus Gründen der Komplexität weggelassen. Die Laufzeitumgebung von Java bietet hierfür im Vergleich zu der schon erwähnten Möglichkeit, Klasseneigenschaften zu erlangen, keine standardisierte Schnittstelle.

Beispiel:

```
import java.util.*;
```

3.5.7 Paket *java.lang*

In der Programmiersprache Java sind alle Klassen des Pakets *java.lang* standardmäßig in der Menge der automatisch importierenten Klassen enthalten. Der Entwickler benötigt für die Referenzierung einer dieser Klassen keine *Import*-Anweisung. Da jedoch wegen der fehlenden

Implementierung der *Wildcards* die Möglichkeit, ganze Pakete zu importieren, nicht bestand, wurden die folgenden Klassen des Paketes *java.lang* fest zur Liste der Import-Deklarationen hinzugefügt:

- java.lang.Integer
- java.lang.String
- java.lang.Boolean
- java.lang.Character

Für eine Referenzierung dieser Klassen ist nun keine Import-Anweisung mehr nötig.

3.5.8 Primitive Datentypen

In der Programmiersprache Java gibt es zwei verschiedene Arten von Datentypen, die primitiven Datentypen und die Objekttypen. Primitive Datentypen sind keine Objekte, besitzen also auch keine objektorientierten Eigenschaften. Für Container-Klassen wie *java.util.Vector* sind primitive Datentypen deshalb nicht geeignet.

Aus diesem Grund sind die primitiven Datentypen für den Einsatz in der Typinferenz eher uninteressant. Bei der Entwicklung des vorhandenen Compilers wurde die Implementierung dieser Datentypen nur rudimentär vorgenommen.

Jeder primitive Datentyp besitzt in Java eine objektorientierte Wrapper-Klasse, die den primitiven Datentyp in einem Objekttyp kapselt.

Bei numerischen und booleschen Operatoren werden im bestehenden Compiler im Gegensatz zur Java-Sprachspezifikation keine primitiven Datentypen sondern deren Wrapper-Klassen erwartet.

Da die Beschränkung der Containerklassen auf Objekttypen für den Entwickler bedeutet, dass er alle primitiven Datentypen stets in ihre Wrapper-Klassen kapseln muss, wurde von Sun für die Version 5.0 von Java das *Autoboxing*-Verfahren entwickelt, welches eine Kapselung der primitiven Datentypen in ihre Wrapper-Klassen automatisch vornimmt.

So könnte man folgende Code-Sequenz mit einem Java-Compiler der Version 1.4 nicht compilieren:

```
Integer i=1;
Integer j=1;
Integer k=i+j;
```

Ein Java 5.0-Compiler führt die benötigte Kapselung von *int* nach *java.lang.Integer* und zurück automatisch durch. Beim hier vorliegenden Compiler wäre nur die oben dargestellte Code-Sequenz gültig. Eine Addition von primitiven Datentypen ist nicht gültig.

Da jedoch alle Klassen, die durch *Import*-Deklarationen in die Menge der Basis-Typannahmen geladen werden, primitive Datentypen beinhalten können, mussten um Typfehler zu vermeiden alle primitiven Datentypen direkt nach dem Einlesen durch ihre Wrapper-Klassen ersetzt werden.

Die Signatur der Methode *Vector.elementAt()* sieht demnach nach dem Ersetzen der primitiven Datentypen folgendermaßen aus:

```
Vector=elementAt: java.lang.Integer --> GenericTypeVar E
```

3.6 Überladene Methoden

In der Programmiersprache Java ist es möglich, mehrere gleichbenannte Methoden in einer Klasse zu definieren, solange sie eine unterschiedliche Signatur besitzen. Die Signatur einer Methode ist die Aufzählung der Typen aller Aufrufparameter, die die Methode besitzt.

```
class Overloaded{
        int  method(int  a){ ... }
        byte method(byte a){ ... }
}
```

Die bisherige Implementierung des TRA kann Methodendefinitionen dieser Art jedoch nicht auflösen. Der Grund hierfür liegt in den Datenstrukturen, in denen nach [Bae05] Deklarationen von Variablen und Methoden abgelegt werden.

Alle Typannahmen werden in HashTables gespeichert, um einen performanten und eindeutigen Zugriff zu ermöglichen. Die Hash-Schlüssel, die für die Eintragung eines Wertes in die Tabelle nötig sind, boten bisher keine Möglichkeit, überladene Methoden korrekt abzuspeichern:

Typannahme	Notation
InstVar	InstVar#Klassenname.Identifier
MethodType	Method#Klassenname.Methodenname(AnzahlParameter)
MethodPara	MethodPara#Klassenname.Methodenname(AnzahlParameter).Identifier
LocalVar.	LocalVar#Klassenname.Methodenname(AnzahlParameter).Block_BlockID.Identifier

Beide Hash-Schlüssel der Methoden aus dem Overloaded-Beispiel würden demnach folgendermaßen aussehen:

```
Method#Overloaded.method(1)
```

Da die Schlüssel identisch sind, wird die erste Methode beim Ablegen in der Hashtable durch die Zweite überschrieben. Die Informationen der ersten Methode gehen dadurch verloren. Sie wird deshalb durch den TRA nicht inferiert. Um diesem Problem entgegenzuwirken, wurden die Hash-Schlüssel um einen Identifier *OverloadedID* erweitert, welcher die Anzahl der Methoden mit gleicher Signatur fortlaufend zählt. Die Schlüssel haben nun folgende Struktur:

Typannahme	Notation
InstVar	InstVar#Klassenname.Identifier
MethodType	Method#Klassenname.Methodenname(AnzahlParameter;OverloadedID)
MethodPara	MethodPara#Klassenname.Methodenname(AnzahlParameter;OverloadedID).Identifier
LocalVar	LocalVar#Klassenname.Methodenname(AnzahlParameter;OverloadedID).Block_BlockID.Identifier

Damit die Hash-Schlüssel einfach verwalt- und erkennbar sind, wurden die Schlüssel-Klassen, die die OverloadedID beinhalten, mit dem Interface *IMethodBoundKey* markiert. Dieses Interface besitzt Methoden, die es dem TRA ermöglichen, vom aktuellen Schlüssel aus durch alle darauffolgenden Schlüssel zu iterieren. Die Methode *getNextMethodKey()* liefert beispielsweise den Hash-Schlüssel der Typannahme, dessen OverloadedID um eins inkrementiert wurde.

Vor dem Einfügen einer Typannahme in die Hashtable CHashTableSet muss nun überprüft werden, ob sich der Schlüssel schon in der Hashtable befindet. Ist dies der Fall, wird durch die Methode *getNextMethodKey()* der nächste Schlüssel generiert, mit dem ein erneuter Versuch gestartet werden kann.

Konnte die Typannahme korrekt abgelegt werden, wird die OverloadedID im Method-Objekt abgelegt.

Wird im zu compilierenden Programm eine Methode aufgerufen, muss die Typannahme dieser Methode aus der HashTable geladen werden. Damit an dieser Stelle mehrere Ergebnisse auftauchen können, wurde die Klasse CHashTableSet um die Methode getElements(IHashSetKey) erweitert. Diese durchsucht die HashTable nach *allen* Methoden, die die angegebene Signatur besitzen.

3.7 Fehlermarkierung in der IDE

Ein wichtiges Element von Compilern ist die Entdeckung und Behandlung von Fehlern im zu compilierenden Programm. Je besser die Fehleranalyse des Compilers ist, desto einfacher gestaltet sich für den Anwender des Compilers die Behebung eines Fehlers im zu compilierenden Programm.

Der TRA kann aus den verschiedensten Gründen die Typrekonstruktion nicht beenden. Falsche Typen bei einer Zuweisung oder ein Aufruf einer nicht vorhandenen Methode wären Beispiele hierfür.

Im Eclipse-Plugin wird bisher nur eine Nachricht auf dem Bildschirm des Anwenders ausgegeben, welche den aufgetretenen Fehler beschreibt. Hat der Anwender ein Programm entwickelt, welches mehr als nur ein Paar Zeilen Quellcode enthält, ist eine genaue Auffindung der Fehlerursache mit einem hohem Aufwand verbunden. Der Anwender bekommt bisher keine Informationen über die Stelle im Quellcode, in der sich der Fehler befindet.

Um dem entgegen zu kommen und somit den Komfort der Benutzeroberfläche zu erhöhen, wurde eine Schnittstelle für die Studienarbeit [Hor06] definiert, durch die es möglich ist, im Eclipse-Plugin die Anweisung, die den aufgetretenen Fehler verursacht hat, zu markieren.
Dazu musste der Typ *CTypeReconstructionException* modifiziert werden. Im Konstruktor dieser Exception, die an jeder Stelle, in der ein semantischer Fehler aufgetreten ist, geworfen wird, kann nun der *CTypeReconstructionException* das Objekt, das das Problem verursacht hat, mitgegeben werden. Anhand dieses Objektes kann im Eclipse-Plugin dessen Offset berechnet und ausgewertet werden. Der Offset einer Anweisung berechnet sich aus der Anzahl der Zei-

chen, die vor der Anweisung - von Beginn des zu compilierenden Programmes an - stehen.

Der Offset selbst kann beim Parsen direkt von den erkannten *Token*-Objekten über deren Methode *getOffset()* erlangt werden. Damit jede Anweisung, Deklaration und Operation, die sich im abstrakten Syntaxbaum des Programmes befindet, die Information über Ihr Offset beinhaltet, mussten alle Klassen, die eine mögliche Ursache für semantische Fehler sind, das Interface *IItemWithOffset* implementieren.

Dieses Interface besitzt die Methode *int getOffset()* und stellt die schon erwähnte Schnittstelle zum Eclipse-Plugin dar. Diese Methode muss von jeder der folgenden Klassen, die das Interface implementieren, ausprogrammiert werden:

- Method
- Operator
- Statement
- Type
- UsedId

Klassen, die Anweisungen, Operatoren oder Konstrukte beinhalten, die keine semantischen Fehler auslösen können, geben *-1* als Offset zurück. Alle anderen Klassen müssen den konkreten Offset beinhalten und zurückgeben.
Um sicher zu stellen, dass diese Klassen, die Information über den Offset besitzen, wurde ein neuer Parameter *Offset* in die Konstruktoren dieser Klassen eingefügt.

4 Fehlerbehebung

4.1 Typunsicherheiten bei Container-Klassen

Der vorliegende Java-Compiler und sein abstrakter Syntaxbaum wurde in der Version 1.4 von Java entwickelt. In dieser Version gab es noch keine generischen Datentypen. Deshalb wurden ursprünglich alle Container-Typen nichtgenerisch instanziert. Seit der Veröffentlichung von Java 1.5 wurden der Compiler und der TRA auf der Basis von generischen Container-Typen weiterentwickelt. Die ursprünglichen nichtgenerischen Container wurden jedoch nicht zu generischen umgewandelt.

Zu Beginn der Studienarbeit warnte die Entwicklungsumgebung vor ca. hundert Typunsicherheiten die von nichtgenerischen Container-Typen verursacht wurden. Durch das korrekte Spezifizieren der Container-Typen konnte die Anzahl der Typunsicherheiten auf ca. zehn minimiert werden. Diese sind aufgrund der Struktur des Parsers (dieser unterstützt keine generischen Datentypen) nicht auflösbar. Beim Spezifizieren der Container-Typen konnten zwei potentielle TypeCast-Fehler gefunden und beseitigt werden. »If my programs have the right types, they are nearly already work also correct« [Plu05].

4.2 GenericTypevars wurden falsch erkannt

In der vorliegenden Implementierung des Compilers werden alle Variablen, die eines generischen Typs sind, als Reference-Type - sprich als Referenz auf ein Objekt - und nicht als generische Typvariable erkannt.

Dies liegt daran, dass der Parser zur Laufzeit noch keine Information besitzt, ob es sich um eine generische Typvariable oder einen Klassennamen handelt.

Es wurde also beispielsweise bei folgender Methode angenommen, dass es sich bei *E* um eine Referenz auf ein Objekt handelt:

```
public E elementAt(int i){...}
```

Um dieses Problems zu lösen, musste jede Instanzvariable und Methode nach RefTypes durchsucht werden. Falls es eine gleichnamige generische Typvariable gibt, wird nun der RefType durch diese ersetzt. Die Suche muss rekursiv auf dem ganzen Baum erfolgen, da sich in jedem Block eine Deklaration einer generischen Typvariable befinden kann.

4.3 Exceptions

Während des Ablaufs des TRA werden viele Typannahmen getroffen. Einige davon müssen im Laufe des Algorithmus wieder aus der Menge der Annahmen entfernt werden, weil andere Anweisungen diese nicht erlauben.

Beispiel:

```
public void method(){
        x;                  // Zeile 1
        x=1;                // Zeile 2
        x++;                // Zeile 3
}
```

In *Zeile 2* kann es sich bei *x* um die Typen *Object, Number oder Integer* handeln. Die Inkrement-Operation in *Zeile 3* kann jedoch nur auf eine Variable vom Typ *Integer* angewandt werden. An dieser Stelle müssen die Typannahmen *Object und Number* aus der Menge der Typannahmen entfernt werden.

Es müssen stets alle Typannahmen für eine Anweisung überprüft werden. In der vorliegenden Implementierung bricht der TRA jedoch bei einer fehlerhaften Unifizierung ab und wirft eine CTypeReconstructionException. Die weiteren Typannahmen werden gar nicht erst geprüft.

Um dieses Problem zu lösen, muss an jeder Stelle im TRA, an der durch eine Menge von Typannahmen iteriert wird, eine potentielle CTypeReconstructionException gefangen werden. Solange es mindestens eine Typannahmen gibt, deren Typrekonstruktion nicht mit einer Exception abbricht, kann der TRA fehlerfrei fortsetzen. Fehlerhafte Typannahmen müssen verworfen werden.

Gibt es keine Typannahme in der Menge der Typannahmen, die erfolgreich geprüft werden konnte, muss der TRA mit einer Exception abbrechen.

Wenn es nur eine Typannahme in der Menge gab und genau diese schief gelaufen ist, kann die von dieser Typannahme verursachte Exception weiter geworfen werden. Die Information über die Zeile, die den Fehler verursacht hat, ist somit nicht verloren gegangen.

Gab es mehrere Typannahmen, von denen alle einen Fehler verursacht haben, müssen alle geworfenen Exceptions gesammelt werden. Diese können nun einer neu erzeugten Exception in einem Vector hinzu gefügt werden. Damit kann später dem Anwender die Ursache des Problems in der IDE genauer angezeigt werden.

4.4 Leere Blöcke

Beinhalten Methoden leere Blöcke, so hatte dies bisher die Folge, dass der TRA wegen eines Fehlers in der Implementierung für diesen Block eine leere Menge an Typannahmen zurückgegeben hat. Alle bisher berechneten Typannahmen anderer Methoden, Klassen und die der BasicAssumptions wurden damit gelöscht und waren somit nicht mehr verfügbar. Suchte der

TRA dann zuvor deklarierte Klassen, Methode, Variablen oder Basis-Klassen, wurde er nicht fündig und brach mit einer Exception ab.
Beispiel:

```
class Example {
        ...
        public leereMethode(){
                // Hier werden alle zuvor berechneten
                //  Typannahmen elimeniert
        }
}
```

Um dieses Problem zu lösen, musste die Typrekonstruktion eines Blocks angepasst werden. Bisher wurde im Falle eines leeren Blocks eine leere Menge von Typannahmen zurückgegeben. Zu dieser Menge musste die bisher berechnete Menge hinzugefügt werden, damit diese nicht verloren geht.

4.5 Rückgabewerte von Methoden

In der vorliegenden Implementierung findet bei der Typrekonstruktion keine Prüfung des Return-Typs statt. Wurde dieser nicht explizit festgelegt, wird er korrekt erkannt und gesetzt. Hat der Anwender diesen jedoch explizit im Programm definiert, führten auch Typen in der Return-Anweisung, die nicht zu dem Return-Typ der Methode passten, zu einem Ergebnis.

Folgende Methode wurde bisher ohne Erkennung eines Fehlers compiliert:

```
public Boolean test(){
        return("TestString");
}
```

Eine weitere Folge dieses Problems wird deutlich:

```
public Integer test(){
        x;                          // Zeile 1
        x=1;                        // Zeile 2
        return(x);                  // Zeile 3
}
```

In *Zeile 2* des letzten Beispiels sind für *x* die Typen *Object, Number oder Integer* möglich. Durch das Return-Statement in *Zeile 3* sollten die hier ungültigen Typannahmen *Object und Number* entfernt werden, da sie sich nicht mit dem Return-Typ der Methode vereinbaren lassen.

Der Grund hierfür ist eine fehlerhafte Implementierung des Algorithmus *TRStart* [Plu05]. Nach der Typrekonstruktion aller Anweisungen der Methode im Programmteil *TRNextMeth* wird der berechnete Rückgabetyp mit dem deklarierten Rückgabetyp der Methode unifiziert. Wurde kein Rückgabetyp deklariert, wird bei der Unifikation eine Substitution des TypePlaceholders auf den berechneten Rückgabetyp erzeugt. Der Return-Typ wurde somit erkannt. Hat

der Anwender explizit einen Rückgabetyp deklariert, wird auch dieser mit dem berechneten Typ unifiziert. Jedoch wird unabhängig vom Ergebnis der Unifikation die aktuelle Typannahme stets in die Ergebnismenge übernommen. Selbst wenn keine Unifikation vom berechneten und dem deklarierten Rückgabetyp möglich ist.

Durch die Korrektur dieses Fehlers konnten die oben aufgeführten Probleme gelöst werden.

4.6 Methodensignatur bei der Codegenerierung

Bei der Generierung des Bytecodes anhand des abstrakten Syntaxbaumes wird bei einem Methodenaufruf die Orginalsignatur der aufgerufenen Methode benötigt. Diese Information liegt jedoch bisher bei der Bytecodegenerierung nicht vor.

Um diese Information der Bytecodegenerierung bereit zu stellen, musste eine genaue Analyse durchgeführt werden, an welcher Stelle Informationen über die aufgerufene Methode vorhanden sind.

Während des Parsens der zu compilierenden Datei ist es noch nicht möglich, die Signatur der aufgerufenen Methode genau zu spezifizieren, da dem Parser nur die bisher eingelesenen Methoden und Klassen bekannt sind. Klassen, die später folgen oder durch Import-Anweisungen in den Compiler geladen werden, sind hier noch nicht bekannt. Beim Erzeugen des MethodCall-Objektes ist es also ausgeschlossen, die Signatur zu erlangen.

Bei der Typrekonstruktion selbst kann man in bestimmten Fällen nicht mit Sicherheit voraussagen, welche Signatur die aufgerufene Methode besitzt. Anhand des folgenden Beispiels soll das Problem genauer erläutert werden:

```
class Worker1{
        void doSomething(String x){...}
}
class Worker2{
        void doSomething(Integer y){...}
}
class Content1{
        String getElem(){...}
}
class Content2{
        Integer getElem(){...}
}
class User{
        meth(worker,content){
                worker.doSomething(content.getElem());
        }
}
```

Bei dem Methodenparameter *content* der Methode *meth* erzeugt der TRA zwei Typannahmen, zwischen denen der Anwender in der GUI wählen kann. Bei dem Parameter kann es sich

entweder um den Typ *Content1* oder *Content2* handeln. Abhängig davon, welchen Typ der Anwender in der GUI für *content* wählt, wird der Typ der Variable *worker* automatisch entweder auf *Worker1* oder auf *Worker2* gesetzt. Die Methode *doSomething* hat in den Klassen *Worker1* und *Worker2* eine unterschiedliche Signatur.

Dementsprechend kann während der Typrekonstruktion nicht eindeutig festgelegt werden, welche Signatur die aufgerufene Methode besitzt. Deshalb wurde der Algorithmus TRMCallApp dahingehend erweitert, dass er bei jeder erfolgreich gefundenen Methode die Signatur in einer Hashtable ablegt. Als Key, unter dem die Signatur abgelegt wird, dient der Klassenname der gefundenen Klasse. Im vorausgegangenen Beispiel würde diese Hashtable nach der Typrekonstruktion folgende Informationen beinhalten:

```
Worker1=doSomething: java.lang.String --> void
Worker2=doSomething: java.lang.Integer --> void
```

Bei der Codegenerierung ist schließlich der Typ des Objektes, auf das die Methode aufgerufen wurde, bekannt. Der Anwender hat sich in der IDE auf den Typ der Variable worker festgelegt. Nun kann die passende Signatur aus der Hashtable mithilfe des Klassennamens entnommen werden.

Bei nicht vererbten Klassen konnte die Signatur durch dieses Vorgehen eindeutig bestimmt werden. Handelt es sich jedoch bei einer der beiden möglichen Klassen um eine Subklasse der anderen, kam ein Problem auf, welches das nachfolgende Beispiel erläutern soll:

```
class Worker1{
        void doSomething(String x){...}
}
class Worker2 extends Worker1{
}
class User{
        meth(worker){
                worker.doSomething("AnyString");
        }
}
```

Der Algorithmus *TRMCallApp* [Plu05] durchsucht, um den Typ der Variable *worker* festzulegen, jede Klasse nach der Methode *doSomething*. Bei der Klasse *Worker1* liefert der TRA ein positives Ergebnis und legt *Worker1* und alle Subklassen - in diesem Fall die Klassen *Worker1* und *Worker2* - in der Menge der Typannahmen ab. Beim Untersuchen der Klasse *Worker2* findet der Algorithmus die Methode *doSomething* nicht und legt diese auch nicht in der Hashtable ab - schließlich hat diese Klasse die Methode *doSomething* nur implizit durch die Vererabungshierarchie erworben.

Wählt der Anwender nun jedoch *Worker2* als Variablentyp von *worker*, so wurde die Methode *doSomething* in der bisherigen Implementierung nicht in der Hashtable mit dem Key *Worker2* gefunden.

Um Methodenaufrufe für Subklassen zu ermöglichen, wird nun beim Ablegen der Methodensignaturen in der Hashtable die Methodensignatur auch für alle Subklassen der gefundenen Klasse abgelegt.

4.7 Konstruktoren

Konstruktoren von Basisklassen wie *Integer*, *String*, etc. werden in der Menge der Methoden wie eine normale Methode, die jedoch den Bezeichner $\langle init \rangle$ hat, abgelegt.

Werden im zu compilierenden Programm selbst Konstruktoren definiert, kann der Parser jedoch nicht erkennen, dass es sich um einen Konstruktor handelt, da der Parser erst beim Erstellen des Class-Objektes - also ganz am Ende des Parsens - weiß, welchen Bezeichner die Klasse hat. Der in der Menge der Methoden abgelegte Konstruktor hat folglich statt des Bezeichners $\langle init \rangle$ den Namen der Klasse.

Bei der Instanzierung einer selbstdefinierten Klasse wird der Konstruktor deshalb nicht in der Menge der Methoden gefunden, wenn nach einer Methode gesucht wird, die $\langle init \rangle$ heißt.

Deshalb wurde der Algorithmus *TRProg* dahingehend modifiziert, dass alle Methoden, die gleich heißen wie die Klasse in der sie definiert sind und keinen Return-Typ angegeben haben in einen Konstruktor umgewandelt werden und somit den Methodennamen $\langle init \rangle$ bekommen.

5 Schlussbetrachtungen

5.1 Zusammenfassung

Im Rahmen dieser Studienarbeit wurden einige Fehler im bestehenden Java-Compiler und dem darauf aufbauenden TRA entdeckt. Typunsicherheiten und Design-Schwächen (→ siehe 3.5) konnten behoben werden, Fehler in essentiellen Java-Strukturen wie Konstruktoren wurden behoben.

Der TRA führt nun die Typerkennung nicht nur bei definierten Use-Cases erfolgreich durch, er ist viel mehr inzwischen so stabil geworden, dass der TRA komplexere, bisher noch nicht getestete Programme fehlerfrei typifizieren kann.

Durch die erweiterte Behandlung und Markierung von Fehlern konnte die Anwendbarkeit des Compilers und des Eclipse-Plugins gesteigert werden.

Der Compiler und der zugehörige TRA wurde zudem auf die Java-Strukturen Interfaces, gebundene *(bounded)* generische Datentypen und Packages erweitert und besitzt einen größeren Funktionsumfang.

Die gleichzeitige Compilierung und Typifizierung mehrerer Klassen wurde in Zusammenarbeit mit den Studenten der Partnerstudienarbeiten mit geringen Einschränkungen ermöglicht.

5.2 Ausblick

Nach wie vor unterstützt der vorliegende Compiler grundlegende Sprachinhalte von Java nicht.

Arrays werden vom Parser nicht erkannt, primitive Datentypen werden wie in → 3.5.8 beschrieben entgegen der Sprachspezifikation von Java verwendet. Die Referenzierung statischer Methoden und Attribute einer Klasse ist bisher nicht möglich. Von den in Java verfügbaren verschiedenen Schleifen wurde nur die *While-Schleife* implementiert.

Fraglich ist jedoch, ob eine volle Unterstützung dieser Funktionen in diesem Projekt wichtig ist. Vielmehr geht es hierbei um die Typinferenz, bei der die oben genannten Sprachelemente hauptsächlich eine untergeordnete Rolle spielen.

Wie im letzten Kapitel erwähnt, gibt es für die gleichzeitige Typerkennung mehrerer Klassen Einschränkungen. Es ist im aktuellen Stand nicht möglich, Methoden und Attribute von Klassen zu referenzieren, die **nach** der Deklaration der aktuellen Klasse folgen.

Es können demnach nur Klassen verwendet werden, die in der *jav*-Datei **über** der Klasse stehen.

Um dieses Problem zu lösen müssen Klassen, die andere Klassen referenzieren, *zusammen* mit diesen Klassen typinferiert werden. Damit diese Beziehungen gefunden werden können, müssen Abhängigkeitsgraphen aufgebaut werden, die diesen Zustand darstellen können.

Eine weitere Funktion, die für die Typrekonstruktion interessant ist, ist die Einführung von *BoundedTypePlaceholder*s und wurde bereits in → 3.3.2 erwähnt. Diese Funktion könnte eine Teilaufgabe einer weiterführenden Studienarbeit sein.

Anhänge

UML-Diagramme der aktuellen Compiler-Version befinden sich auf dem beigefügten Datenträger und im CVS-System unter `/doc/UML/`.

Abkürkungsverzeichnis

API	Application Programming Interface
CVS	Concurrent Versions System
GUI	Graphical User Interface
IDE	Integrated Development Environment
JAV	Dateityp des Typinferenz-Compilers
JRE	Java Runtime Environment
JVM	Java Virtual Machine
log4j	Logging-Framework für Java
TRA	Typrekonstruktionsalgorithmus

Literaturverzeichnis

[Bae05] Joerg Baeuerle. *Typinferenz in Java, Studienarbeit*. Berufsakademie Horb, 2005.

[BS01] Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*, chapter 8, pages 447–533. 2001.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[Haa04] Markus Haas. *Weiterentwicklung der Java-Codegenerierung zur Ausfuehrung von parametrisierten Datentypen, Studienarbeit*. Berufsakademie Horb, 2004.

[Hor06] Thomas Hornberger. *Typinferenz in Java - IDE, Studienarbeit*. Berufsakademie Horb, 2006.

[KR90] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall software series. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1990.

[Mel05] Markus Melzer. *Integration der Java-Typinferenz in eine Programmier-Umgebung, Studienarbeit*. Berufsakademie Horb, 2005.

[Ott04] Thomas Ott. *Typinferenz in Java, Studienarbeit*. Berufsakademie Horb, 2004.

[Plu05] Martin Pluemicke. *Type Inference in Generic Java, not yet published*. Berufsakademie Horb, 2005.

[Rei03] Felix Reichenbach. *Erweiterung der semantischen Analyse des Java-Compilers, Studienarbeit*. Berufsakademie Horb, 2003.

[uJB06] Martin Pluemicke und Joerg Baeuerle. *Typeless programming in Java 5.0*. Berufsakademie Horb, 2006.

[Wik06] Wikipedia. *Die freie Enzyklopaedie*. http://de.wikipedia.org/, 2006.