Erweiterte Typinferenz in Java 5.0

Studienarbeit
während den Semestern 5 und 6

im
Studiengang
Informationstechnik

an der
Berufsakademie Stuttgart

vorgelegt von
Achim Burger

Ausbildungsbetrieb
Rechenzentrum der Universität Stuttgart

Stuttgart, Juni 2007

Erklärung

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

Erweiterte Typinferenz in Java 5.0

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

Stuttgart, 29.06.07

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Christian Ullenboom, Java ist auch eine Insel, Galileo Computing, 5. Auflage 2006, ISBN: 3-89842-747-1

[Plu04] Martin Plümicke, Typinferenz in Java, BA Horb 2005

Internetseiten (gesichtet am 29.06.07):

[1] jay – Ein yacc für Java

http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/staff/design/de/Artikel.htmld/index.html

[2] The Java Language Specification, Third Edition

http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

Wikipedia – Die freie Enzyklopädie

http://de.wikipedia.org/

Studienarbeiten:

[Rei03] Felix Reichenbach, Studienarbeit BA Horb 2003
[Haa04] Markus Haas, Studienarbeit BA Horb 2004
[Ott04] Thomas Ott, Studienarbeit BA Horb 2004
[Bae05] Jörg Bäuerle, Studienarbeit BA Horb 2005
[Mel05] Markus Melzer, Studienarbeit BA Horb 2005
[Sch06] Jürgen Schmiing, Studienarbeit BA Horb 2006
[Hor06] Thomas Hornberger, Studienarbeit BA Horb 2006
[Hol06] Timo Holzherr, Studienarbeit BA Horb 2006

Inhaltsangabe

Die Einleitung behandelt die Frage, warum eine Notwendigkeit für einen Compiler wie den vorliegenden besteht, der nicht angegebene Typen aus dem Kontext berechnen kann. Weiterhin ist in diesem Abschnitt die Aufgabenstellung für die Studienarbeit formuliert.

Das Kapitel „Umgebung“ beschreibt, welche Werkzeuge bei der Studienarbeit zum Einsatz kommen sowie welche theoretischen Grundlagen notwendig sind.

Das folgende Kapitel „Benutzung des Compilers“ erklärt, wie der Compiler verwendet wird, welche Eingaben nötig sind und welche Ausgaben zu erwarten sind.

In Kapitel 3 wird untersucht, wo die Funktionalität des Compilers Lücken aufweist. Es werden reservierte Wörter, Operatoren und Konstrukte der Sprache Java getestet.

Das Kapitel 4 beschäftigt sich mit der Implementierung und beschreibt, welche Methoden implementiert werden müssen und wie die Grammatik erweitert wird.

Kapitel 5 liefert ein Fazit der Studienarbeit und gibt einen Ausblick auf die Arbeiten, die noch zu erledigen sind.

Der Anhang enthält den erarbeiteten Programmcode und den Teil der Grammatik, der hinzugefügt wurde.

1 Einleitung

In der Programmiersprache Java war es bisher erforderlich, die Datentypen von Variablen sowie von Rückgabewerten und Attributen der Methoden anzugeben. Häufig sind diese Angaben aber überflüssig, weil sie sich aus dem Kontext berechnen lassen.

Diese Studienarbeit analysiert einen bestehenden Mini-Java-Compiler und erweitert ihn in Richtung der Java 5.0 Vollversion.

1.1 Motivation

Die bei Java notwendigen Typangaben können insbesondere bei größeren Konstrukten recht umständlich werden. Abhilfe soll ein Plugin für die Entwicklungsumgebung Eclipse schaffen, das den von Martin Plümicke in [Plu04] spezifizierten Typinferenz-Algorithmus implementiert.

Das Compilerprojekt wurde im Rahmen zweier Vorlesungen des Jahrganges TIT 2000 an der BA Horb angestoßen. Bisher befassten sich schon mehrere Studienarbeiten mit dem Thema: [Rei03, Haa04, Ott04, Bae05, Mel05, Sch06, Hor06 und Hol06].

Aktuell unterstützt der Compiler allerdings lediglich Mini-Java, eine Teilmenge der Java 5.0 Grammatik. Der Funktionsumfang soll nun in Richtung der Vollversion Java 5.0 erweitert werden, um den Compiler sinnvoll nutzen zu können.

1.2 Aufgabe

In den Vorgängerstudienarbeiten [hoti06, hoth06, scju06, otth04] wurde ein Typinferenz-Algorithmus prototypisch implementiert. Darüber hinaus wurde der Algorithmus ebenfalls in Studienarbeiten in ein Eclipse-Plugin integriert.

In dieser Studienarbeit soll das bisher implementierte Mini-Java zu einer Java 5.0 Vollversion ergänzt werden.

Hierfür sind folgende Arbeitsschritte notwendig:

- Festlegung, welche Features implementiert werden müssen
- Anpassung des Parsers und der abstrakten Syntax
- Erarbeiten und implementieren des Typinferenz-Algorithmus für die neuen Features
- Implementierung der Bytecodeerzeugung für die neuen Features

2 Umgebung

2.1 Eingesetzte Software

Das Compiler-Plugin wurde für Eclipse 3.1.1 entwickelt. Auf einer neueren Version von Eclipse traten Probleme auf.

Als Entwicklungsumgebung für die Features, die im Rahmen dieser Studienarbeit implementiert werden, wird ebenfalls Eclipse 3.1.1 verwendet. Als Compiler und Laufzeitumgebung kommt Java SE 5.0 zum Einsatz.

Zur Versionsverwaltung, Sicherung und für den Austausch von Dateien wird ein CVS auf dem Server „herbie.ba-horb.de“ verwendet. Die beiden benötigten Projekte „JavaCompilerCore“ und „JavaCompilerPlugin“ liegen dort unter dem Pfad „/bahome/projekt/cvs“.

Zur Überprüfung der erzeugten class-Files kommt die Freeware „Cavaj Java Decompiler 1.11“ zum Einsatz.

2.2 Erstellung des Parsers

Für die Generierung des Parsers wird das Programm jay verwendet, eine weitgehend originalgetreue Variante des UNIX-Werkzeugs yacc, das sich aber laut [1] durch verbesserte Fehlermeldungen und die Möglichkeit, eine Ablaufverfolgung generieren zu können, auszeichnet.

```
%token <node> NUMBER
%type  <node> expr
%left  '+' '-'
%left  '*' '/' '%'
%%
expr : expr '+' expr { $$ = newAdd($1, $3); }
| expr '-' expr { $$ = newSub($1, $3); }
| expr '*' expr { $$ = newMul($1, $3); }
| expr '/' expr { $$ = newDiv($1, $3); }
| expr '%' expr { $$ = newMod($1, $3); }
| NUMBER
```

Abbildung 1: Quelle zur Konstruktion eines Formelbaums

2.3 Theoretische Grundlagen

2.3.1 Die Programmiersprache Java

Java ist eine objektorientierte Programmiersprache mit C-ähnlicher Syntax [2]. Der Compiler übersetzt Java-Programme in plattformunabhängigen Bytecode, der wiederum von einer plattformabhängigen Java Virtual Machine (einer Laufzeitumgebung) interpretiert. Im Unterschied zu anderen interpretierten Sprachen wird zur Laufzeit von der Virtual Machine plattformabhängiger Maschinencode erzeugt. Damit entgeht man den Geschwindigkeitseinbußen, die Interpretersprachen üblicherweise mit sich bringen.

Java ist universell einsetzbar auf Systemen, für die eine Virtual Machine dafür existiert. Dies sind PCs mit unterschiedlichen Betriebssystemen, PDAs und Handys. Sogar spezielle Mikroprozessoren, die Java-Bytecode nativ ausführen können, gibt es bereits. Durch die Objektorientierung ist eine hohe Wiederverwendbarkeit von Softwaremodulen möglich.

2.3.2 Compiler

Nach [Bae05] ist ein Compiler ein Programm, das ein Computerprogramm von seiner Quellsprache in ein semantisch entsprechendes Programm einer Zielsprache übersetzt. Die Quellsprache ist hierbei meist eine höhere Programmiersprache, die Zielsprache dagegen eine hardwarenahe Sprache wie Assembler, Bytecode oder Maschinensprache. Des Weiteren legt ein Compiler Symboltabellen an, in denen er Informationen über die im Quellcode verwendeten Bezeichner speichert. Dadurch können Fehler erkannt und gemeldet werden.

Der gesamte Vorgang wird „Kompilieren“ genannt und setzt sich aus zwei Phasen zusammen:

In der Analysephase wird die hierarchische Struktur in einen Syntaxbaum umgesetzt und danach der lexikalischen Analyse (der Scanner oder Lexer zerlegt den Quelltext nach den Regeln der Grammatik in Tokens), der syntaktischen Analyse (hier wird überprüft, ob der Quellcode formal richtig ist, also der Grammatik der Quellsprache entspricht) sowie der semantischen Analyse (Überprüfung von Variablen, Zuweisungen und Operationen auf Typkorrektheit) unterzogen.

In der Synthesephase wird aus dem in der Analysephase aufgebauten Syntaxbaum Programmcode der Zielsprache generiert. Die Synthesephase umfasst ihrerseits die Phase der Codeoptimierung, in der versucht wird, Laufzeit und Speicherbedarf des Programms zu verringern, und die Phase der Codegenerierung, die für gewöhnlich letzte Compilerphase, die den endgültigen Programmcode in der Zielsprache erzeugt. Falls die Zielsprache die Maschinensprache ist, kann hier schon ein ausführbares Programm vorliegen, im Fall des Java-Compilers erhält man Bytecode, der von der Java Virtual Machine interpretiert werden kann.

2.3.3 Typinferenz

Mit Typinferenz ist der Rückschluss auf den Typ eines Ausdrucks in der Informatik gemeint, der nicht ausdrücklich angegeben ist. Laut [Hol06] werden Typinferenzalgorithmen als Teil der semantischen Analyse des Compilers direkt auf den abstrakten Syntaxbaum angewandt, um die Typrekonstruktion durchzuführen.

Im vorliegenden Compiler wurde der von [Plu05] definierte Typinferenzalgorithmus im Rahmen der Studienarbeit [Bae05] implementiert, der mithilfe seiner Unteralgorithmen die Typen basierend auf dem Unifikationsalgorithmus aus [Ott04] rekonstruiert.

Typpaare, die unifiziert werden sollen, werden laut [Hol06] anhand von Schlussregeln untersucht. Für die Platzhalter (TypePlaceholders) werden Substitutionen berechnet, durch die sie ersetzt werden.

2.4 Benutzung des Compilers

2.4.1 Typrekonstruktion und Bytecodegenerierung

Die Grammatik befindet sich in der Datei „JavaParser.jay“. Aus ihr wird bei jedem Build mithilfe der Batch-Datei „RunJay.bat“ die Datei „JavaParser.java“ generiert.

Die Typrekonstruktion sowie die Bytecodegenerierung wird mit der main-Methode der Klasse CinferenceTest durchgeführt. Als einzigen Parameter bekommt die Methode bei Aufruf den Pfad zur Datei mit dem Quellcode übergeben. Die Typannahmen, die das Programm nach und nach trifft, werden auf der Konsole ausgegeben:

2.4.2 Ausgaben

2.4.2.1 Erfolgsmeldung Typrekonstruktion

Neben der class-Datei mit dem Bytecode, die im Ordner „Bytecode“ erstellt wird, gibt der Compiler noch einige Fehler- und Erfolgsmeldungen aus.

Die Typen, mit denen die Type Place Holder (TPH) ersetzt werden (Substitutions) und die Typannahmen für Variablen und Methoden-Parameter (Assumptions):

```
Substitutions:
Set {
TPH B --> java.lang.Integer,
TPH A --> java.lang.Integer,
TPH C --> java.lang.Integer,
TPH F --> java.lang.String,
TPH E --> java.lang.String,
TPH G --> java.lang.String,
TPH H --> java.lang.String,
TPH D --> java.lang.String
}
Assumptions:
Set {
methode3: NOPARAS --> TPH I,
methode4: NOPARAS --> TPH M,
i: java.lang.Integer,
methode2: NOPARAS --> java.lang.String,
s: int,
methode1: NOPARAS --> void,
text: java.lang.String
}
```

Abbildung 2: Ausgabe des Typinferenzalgorithmus

2.4.2.2 Erfolgsmeldung Codegenerierung

Diese Meldung wird auf der Konsole ausgegeben, wenn die Codegenerierung erfolgreich abgeschlossen wird.

```
 MyCompiler      INFO  [codegen  ] Beginn der Codegenerierung ...
 SourceFile      INFO  [codegen  ] Anzahl der Interfaces: 0
 SourceFile      INFO  [codegen  ] Anzahl der Klassen: 1
 ClassFile       INFO  [codegen  ] Generieren der Klasse: Modifier2
 ClassFile       INFO  [codegen  ] Verarbeite Konstanten-Pool: 31
 ClassFile       INFO  [codegen  ] Verarbeite Interfaces: 0
 ClassFile       INFO  [codegen  ] Verarbeite Fields: 0
 ClassFile       INFO  [codegen  ] Verarbeite Methodenliste: 5
 ClassFile       INFO  [codegen  ] Verarbeite Attribute: 0
 Class           INFO  [codegen  ] Compilierung erfolgreich
 abgeschlossen, Modifier2.class erstellt.
 MyCompiler      INFO  [codegen  ] Codegenerierung beendet!
```

Abbildung 3: Ausgabe Codegenerator

2.4.2.3 Fehlermeldung Lexikalische Analyse

Fehlermeldung, wenn die Lexikalische Analyse fehlschlägt. D. h., wenn unerlaubte Schlüsselwörter, Bezeichner, Zahlen oder Operatoren verwendet werden:

Fehler im Code:

```
class LexTest{
 test(){
  i;
  i = @1;
 }
}
```

Abbildung 4: Fehlertest Lexikalische Analyse

Fehlermeldung:

```
Exception in thread "main" java.lang.Error: Lexical Error: Unmatched
Input.
      at mycompiler.myparser.JavaLexer.yylex(JavaLexer.java:1097)
      at mycompiler.myparser.Scanner.advance(Scanner.java:40)
      at mycompiler.myparser.JavaParser.yyparse(JavaParser.java:669)
      at mycompiler.MyCompiler.parse(MyCompiler.java:267)
      at mycompiler.MyCompiler.parse(MyCompiler.java:500)
      at mycompiler.mytest.CInferenceTest.main(CInferenceTest.java:47)
```

Abbildung 5: Fehlermeldung Lexikalische Analyse

2.4.2.4 Fehlermeldung Syntaktische Analyse

Fehlermeldung, wenn die Syntaktische Analyse fehlschlägt. D. h., wenn der eingelesene Quelltext nicht der Grammatik (Syntax) der Quellsprache entspricht:

Fehler im Code:

```
class SyntaxTest{
  test(){
  i;
  else{
   i = 1;
  }
 }
}
```

Abbildung 6: Fehlertest Syntaktische Analyse

Fehlermeldung:

```
mycompiler.myparser.JavaParser$yyException: syntax error, expecting '('
';' '{' '}' BOOLEAN CHAR FOR IF INT RETURN THIS WHILE INTLITERAL
BOOLLITERAL JNULL CHARLITERAL STRINGLITERAL IDENTIFIER INCREMENT
DECREMENT
      at mycompiler.myparser.JavaParser.yyparse(JavaParser.java:708)
      at mycompiler.MyCompiler.parse(MyCompiler.java:267)
      at mycompiler.MyCompiler.parse(MyCompiler.java:500)
      at mycompiler.mytest.CInferenceTest.main(CInferenceTest.java:47)
```

Abbildung 7: Fehlermeldung Syntaktische Analyse

2.4.2.5 Fehlermeldung Semantische Analyse

Fehlermeldung, wenn die Semantische Analyse scheitert. D. h., wenn der eingelesene Quelltext zwar der Grammatik der Quellsprache entspricht, aber die darüber hinaus gehenden Bedingungen der Sprache an ein Programm nicht erfüllt. Dies tritt beispielsweise auf, wenn eine nicht vorher deklarierte Variable verwendet wird.

Fehler im Code:

```
class Fehlertest{
  test(){
  i = 1;
 }
}
```

Abbildung 8: Fehlertest Semantische Analyse

Fehlermeldung:

```
Assign.TRExp(): Typen i und java.lang.Integer lassen sich nicht
unifizieren.
      at mycompiler.mystatement.Assign.TRExp(Assign.java:370)
      at mycompiler.mystatement.Assign.TRStatement(Assign.java:397)
      at mycompiler.mystatement.Block.TRStatements(Block.java:241)
      at mycompiler.mystatement.Block.TRStatement(Block.java:197)
      at mycompiler.myclass.Class.TRNextMeth(Class.java:1122)
      at mycompiler.myclass.Class.TRStart(Class.java:897)
      at mycompiler.myclass.Class.TRProg(Class.java:789)
      at mycompiler.SourceFile.typeReconstruction(SourceFile.java:753)
      at mycompiler.MyCompiler.typeReconstruction(MyCompiler.java:549)
      at mycompiler.mytest.CInferenceTest.main(CInferenceTest.java:56)
```

Abbildung 9: Fehlermeldung Semantische Analyse

3 Festlegung, welche Features fehlen

Momentan unterstützt der Java-Compiler lediglich eine Mini-Java-Grammatik, eine abgespeckte Version der Java-Grammatik. Um den Compiler soweit erweitern zu können, dass er möglichst das Java 5.0-Sprachspektrum unterstützt, wurden sämtliche reservierten Wörter, Operatoren und einige besondere Konstrukte von Java in Use-Cases gefasst und dem Compiler übergeben. In den Fällen, die reibungslos verlaufen, steht in den Tabellen „ja“, falls Fehlermeldungen bei Typrekonstruktion und/oder Codegenerierung ausgegeben werden „nein“.

3.1 Funktionstest der reservierten Wörter von Java 5.0:

Sprachelement	Ergebnis des Funktionstests
abstract	ja
assert	nein
boolean	ja
break	nein
byte	nein
case	nein
catch	nein
char	ja
class	ja
const	1)
continue	nein
default	nein
do	nein
double	nein
else	ja
enum	nein
extends	ja
false	ja
final	ja
finally	nein
float	nein
for	ja
goto	2)
if	ja
implements	ja
import	ja
instanceof	nein

Sprachelement	Ergebnis des Funktionstests
Int	ja
interface	ja
long	nein
native	nein, wird für einen Returntyp gehalten
new	ja
null	ja
package	ja
private	ja
protected	ja
public	ja
return	ja
short	nein
static	ja
strictfp	nein
super	nein
switch	nein
synchronized	nein
this	nein
throw	nein
throws	nein
transient	nein
true	ja
try	nein
void	ja
volatile	nein
while	ja

Tabelle 1: Funktionstest der reservierten Wörter

[1] In C++ gibt es für Parameter den Zusatz const, an dem der Compiler erkennen kann, dass Objektzustände nicht verändert werden sollen. Ein Programm nennt sich »const-korrekt«, wenn es niemals ein konstantes Objekt verändert. Dieses const ist in C++ eine Erweiterung des Objekttyps, die es in Java nicht gibt. Zwar haben die Java-Entwickler das Schlüsselwort const reserviert, doch genutzt wird es bisher nicht.

[2] Das Schlüsselwort goto wird in Java derzeit (Version 5.0) nicht verwendet. Es ist lediglich reserviert, damit der Compiler für Umsteiger anderer Programmiersprachen sinnvollere Fehlermeldungen ausgeben kann.

3.2 Funktionstest der Operatoren:

Operator	Ergebnis des Funktionstests
=	Ja
>	Ja
<	Ja
!	Ja
~	Nein
?	Nein
:	Nein
==	Nein
<=	Ja
>=	Ja
!=	Ja
&&	Ja
||	Ja
++	Ja
--	Ja
+	Ja [3]
-	Ja
*	Ja
/	Ja

Operator	Ergebnis des Funktionstests
&	Ja
|	Ja
^	Ja
%	Ja
<<	Nein
>>	Nein
>>>	Nein
+=	Nein
-=	Nein
*=	Nein
/=	Nein
&=	Nein
|=	Nein
^=	Nein
%=	Nein
<<=	Nein
>>=	Nein
>>>=	Nein

Tabelle 2: Funktionstest der Operatoren

[3] Der Plus-Operator funktioniert nicht bei Strings.

3.3 Funktionstest einiger Konstrukte

Sinn und Zweck der Evaluierung der Funktionalität des Compilers ist, aufzuzeigen, welche Features noch nicht funktionieren und implementiert werden müssen. Daher werden im Folgenden diejenigen Konstrukte aufgelistet, die getestet wurden und einen Fehler erzeugten. Die Evaluierung soll hier nicht aufzeigen, welche Konstrukte der Compiler bereits beherrscht.

Die nachfolgend aufgelisteten Fehler traten

- In Java können mehrere Variablen durch Komma getrennt deklariert werden:

  ```
  int variable1, variable2, variable3;
  ```

  Momentan unterstützt dies der Compiler nicht. Es wird stets nur die erste Variable erzeugt.

- Arrays werden – so wie in den folgenden Use-Cases aufgeführt – nicht unterstützt:

  - ```
    class Arraytest1{
      test(){
        int[] array;
        array = new int[10]
      }
    }
    ```

    Ausgegebene Fehlermeldung: `syntax error, expecting IDENTIFIER`

  - ```
    class Arraytest2{
      test(){
        int[] array;
        array = {1,2,3};
      }
    }
    ```

    Ausgegebene Fehlermeldung: `syntax error, expecting '!' '(' '+' '-' NEW THIS INTLITERAL BOOLLITERAL JNULL CHARLITERAL STRINGLITERAL IDENTIFIER INCREMENT DECREMENT`

  - Auch die folgenden Varianten der Array-Deklaration akzeptiert der Compiler nicht:

    ```
    int array[];

    array[];
    ```

- Bei Methodenaufrufen und der Verwendung von „`return`“ tritt bei der Codegenerierung ein Fehler auf. Die Analyse verläuft dagegen fehlerfrei. Der Compiler lehnt Expressions in Verbindung mit „`return`“ ab, so wie im folgenden Use-Case gezeigt wird. Methodenaufrufe, ob mit oder ohne Parameter, ergeben denselben Fehler:

```
class Test{
  test(){
    a;
    a = 42;
    i;
    i = neg(a * 2);
  }
  neg(i){
    return i*(-1);
  }
}
```

Ausgegebene Fehlermeldung: `NullPointerException`

- Von den möglichen Zahlentypen unterstützt der Compiler derzeit lediglich positive und negative ganze Zahlen, die vom Typrekonstruktionsalgorithmus mit dem Typ Number oder Integer verknüpft werden. Kommazahlen, die Angaben für float und long (4.2f bzw. 42l) und den Präfixen für oktale und hexadezimale Zahlen (0 bzw. 0x) führen zu Fehlern.

  - `4.2f` und `42l` ergeben den Fehler: `syntax error, expecting IDENTIFIER`

  - `0xAB` ergibt den Fehler:
    `java.lang.NumberFormatException: For input string: "0xAB"`

  - `042 wird nicht als oktale Zahl erkannt, sondern als dezimale.`

- Folgende mathematische Operationen führen zu Fehlern bei der Codegenerierung. Hier beispielhaft mit dem Operator *. Der Fehler tritt aber mit den Operatoren +, -, / und % genauso auf:

  - ```
    a;
    b;
    a = 42;
    b = 2*a;
    ```

    Ausgegebene Fehlermeldung:
    `mycompiler.myexception.JVMCodeException: JVMCodeException: Binary: void codegen(ClassFile classfile, Code_attribute code)`

  - ```
    a;
    b;
    a = 42;
    b = a*2;
    ```

    Ausgegebene Fehlermeldung:
    `mycompiler.myexception.JVMCodeException: JVMCodeException: jvmCode: Byte n2n(String s1, String s2)`

4 Implementierung

4.1 Erweiterung der Grammatik

Das For-Statement setzt sich aus vier Variablen zusammen:

- Der Loop-Initializer vom Typ expression, der beim Beginn der Schleife einmal ausgeführt wird.
- Die Loop-Condition vom Typ expression, die vor jedem Durchlauf der Schleife bestimmt, ob der Block in der Schleife ausgeführt wird.
- Die Loop-Expression, die nach jedem Schleifendurchlauf ausgeführt wird und üblicherweise die Zählvariable inkrementiert oder dekrementiert.
- Der Loop-Block vom Typ statement, der den Code enthält, der bei jedem Schleifendurchlauf ausgeführt wird.

Unter der Voraussetzung, dass die Existenz des Loop-Blocks obligatorisch ist, bleiben noch acht Kombinationen des gesamten Ausdrucks, denn die drei Expressions im Schleifenkopf sind alle optional.

Unter dem Eintrag „forstatement“ in der Grammatik steht also für den Fall, dass alle Optionalen Teile vorhanden sind, folgende Zeile mit einem regulären Ausdruck:

```
: FOR '(' expression ';' expression ';' expression ')' statement
```
Abbildung 10: Regulärer Ausdruck zur Erkennung der For-Schleife

```
{     ForStmt Fst = new ForStmt($3.getOffset(),$3.getVariableLength());
      Fst.set_head_Initializer($3);
      Fst.set_head_Condition($5);
      Fst.set_head_Loop_expr($7);
      Fst.set_body_Loop_block($9);
      //Typannahme
      $$ = Fst;
}
```
Abbildung 11: Auszuführender Code, falls der reguläre Ausdruck matcht

Darunter kommt Java-Code, der ausgeführt wird, falls der Quelltext auf den regulären Ausdruck gematcht werden kann:

Hier wird also zuerst ein neues Objekt vom Typ ForStmt instanziiert und dem Konstruktor die Position der For-Schleife im Quelltext sowie deren Ausdehnung übergeben.

Mit den Methoden der Klasse ForStmt werden die drei Expressions und das Statement in Variablen geschrieben. Auf die Teile des Quellcodes wird mit „$x“ zugegriffen, wobei „x“ für die Nummer des Symbols steht. Die Zählung beginnt hier bei 1 und die Symbole sind jeweils durch Leerzeichen getrennt.

Mit der Zeile „$$ = Fst“ wird das Symbol an die nächsthöhere Ebene in der Hierarchie übergeben.

4.2 Implementierung der Klasse ForStmt

4.2.1 Methoden in der Klasse ForStmt

Das Gegenstück zu dem der Grammatik hinzugefügten Abschnitt für die For-Schleife ist die Klasse ForStmt, in der die eigentliche Funktionalität implementiert wird. Die Methoden, die diese Klasse enthält, kommen zum Einsatz, nachdem eine For-Schleife im Quelltext von der Grammatik als solche identifiziert worden ist und führen die Typberechnung und in einer zukünftigen Version auch die Codegenerierung durch.

Die folgenden Methoden sind hier zu implementieren:

- `public ForStmt(int offset, int variableLength)`

  Der Konstruktor der Klasse ruft lediglich den Superkonstruktor (Konstruktor der übergeordneten Klasse „Statement“) auf und übergibt diesem die beiden Parameter „offset“ und „variableLength“, die er selbst erhalten hat. Der Parameter „offset“ gibt die Stelle im Quelltext an, an der sich das Statement befindet, das gerade verarbeitet wird. Der Parameter „variableLength“ enthält die Ausdehnung des Statements im Quelltext. Mithilfe dieser Informationen kann der Compiler präzisere Fehlermeldungen ausgeben.

- `void sc_check(Vector<Class> classname, Hashtable ch, Hashtable bh, boolean ext, Hashtable parach, Hashtable parabh)`

  Diese Methode wurde in früheren Versionen des Compilers benötigt und diente dem Semantic Check, den Browser für gewöhnlich besitzen. Diese Funktion wird aber größtenteils bereits vom Typinferenz-Algorithmus abgedeckt und ist deshalb hier nicht mehr zu implementieren.

- `public void codegen(ClassFile classfile, CodeAttribute code, Vector paralist)`

  und

  `public void loop_codegen(ClassFile classfile, CodeAttribute code, int breakpoint, boolean not, Vector paralist)`

  Diese Methoden wurden derzeit noch nicht in der Klasse ForStmt implementiert. Sie dienen der Codegenerierung, d. h. sie sorgen dafür, dass die For-Schleife im vom Compiler erzeugten class-File enthalten ist.

- `public CTripleSet TRStatement(CSubstitutionSet sigma, CTypeAssumptionSet V, CSupportData supportData)`

  Diese Methode implementiert den Typinferenz-Algorithmus von Martin Plümicke, wie in [Plu04] beschrieben. Das Vorgehen wird im folgenden Abschnitt erläutert.

- `public void wandleRefTypeAttributes2GenericAttributes(Vector<Type> paralist, Vector<GenericTypeVar> genericMethodParameters)`

  Diese Methode wird beim Parsevorgang rekursiv für alle Statements aufgerufen

und passt eine Variable an, falls der Typ generisch ist.

- `public void set_head_Initializer(Expr expr),`

  `public void set_head_Condition(Expr expr),`

  `public void set_head_Loop_expr(Expr expr) und`

  `public void set_body_Loop_block(Statement statement)`

  Diese vier Methoden sind in der Grammatik aufgeführt und werden beim parsen der For-Schleife aufgerufen. Mit ihnen werden die Teile (Initializer, Condition, Loop-Expression und Loop-Block) der For-Schleife in private Variablen der Klasse ForStmt geschrieben. Sie sind also simple Setter.

- `public boolean addOffsetsToStatement(CTypeAssumption localAssumption, String NameVariable, boolean isMemberVariable)`

  Diese Methode gibt die Parameter, die sie beim Aufruf für das Statement ForStmt erhält an den Loop-Initializer und den Loop-Block der For-Schleife weiter, indem sie für diese Elemente wieder die entsprechende Methode `addOffsetsToStatement()` aufruft.

4.2.2 Implementierung des Typrekonstruktions-Algorithmus

Um die verschiedenen Varianten der For-Schleife mit teilweise oder komplett fehlenden Expressions im Schleifenkopf zu unterstützen, wird anfangs überprüft, ob mindestens eine Expression vorhanden ist. Es wird daher von jeder Expression eine Kopie angelegt und mit einer existierenden Expression überschrieben, falls sie leer ist. Das Ergebnis der Typrekonstruktion sollte dadurch nicht verändert werden. Der Code für die Typrekonstruktion kann aber so einfacher gehalten werden. Lediglich die Unifizierung der Condition mit Boolean muss ausgelassen werden, falls die Condition leer ist. Fehlen alle drei Expressions, wird die Typrekonstruktion lediglich für den Loop-Block durchgeführt.

Sind alle Expressions vorhanden, wird die Typrekonstruktion zuerst für den Initializer aufgerufen und die Typannahmen in einem CTripleSet gespeichert. Über diesen Vektor wird nachfolgend iteriert und mit jeder Typannahme die Typrekonstruktion der Condition aufgerufen. Über den Ergebnisvektor hiervon wird abermals iteriert und der Typ jeder Annahme mit Boolean unifiziert. Die Typannahmen der verschiedenen Möglichkeiten, die beim Unifizieren gefunden werden, dienen als Übergabeparameter bei der Typrekonstruktion der Loop-Expression. Und zuletzt wird auch über die Reihe der jetzt getroffenen Typannahmen iteriert und mit jedem Ergebnis-Tripel die Typrekonstruktion des Schleifen-Blocks durchgeführt. Auch die Annahmen, die hier getroffen werden, werden in einem Ergebnis-Set gesammelt und an die aufrufende Instanz zurückgegeben. Damit ist die Typberechnung für das For-Statement abgeschlossen.

5 Fazit und Ausblick

Das Projekt des typenlosen Java-Compilers hat inzwischen gewaltige Ausmaße und ist alles andere als leicht zu durchschauen. Für Novizen auf dem Gebiet Compilerbau ist der Weg der Einarbeitung in das Thema mitunter sehr beschwerlich, auch weil wenig hilfreiche Kommentare im Quelltext zu finden sind. Um nur eine Kleinigkeit zu implementieren, ist ein großer Zeitaufwand erforderlich, um die Grundlagen dafür zu erarbeiten. Das bringt macht Arbeit am Projekt sehr schleppend.

Die folgenden Aufgaben müssen noch erledigt werden:

- Die Codegenerierung für die Klasse ForStmt ist noch zu implementieren
- Die Typrekonstruktion für die For-Schleife funktioniert bei sämtlichen Varianten mit Condition. Hier muss noch nachgebessert werden
- Die seit Java 5.0 mögliche Variante der For-Schleife `for (int i : a)`, wobei a ein Array darstellt, wird derzeit vom Compiler nicht unterstützt.
- Einige reservierte Wörter (Kapitel 3.1), Operatoren (Kapitel 3.2) und Konstrukte (Kapitel 3.3), die in Java erlaubt sind, funktionieren noch nicht. Diese Features müssen noch implementiert werden.

6 Anhang

ForStmt.java

```
package mycompiler.mystatement;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import mycompiler.mybytecode.ClassFile;
import mycompiler.mybytecode.CodeAttribute;
import mycompiler.mybytecode.JVMCode;
import mycompiler.myclass.Class;
import mycompiler.myexception.CTypeReconstructionException;
import mycompiler.myexception.JVMCodeException;
import mycompiler.myexception.SCExcept;
import mycompiler.myexception.SCStatementException;
import mycompiler.myoperator.LogOp;
import mycompiler.myoperator.Operator;
import mycompiler.myoperator.RelOp;
import mycompiler.mytype.GenericTypeVar;
import mycompiler.mytype.Pair;
import mycompiler.mytype.RefType;
import mycompiler.mytype.Type;
import mycompiler.mytypereconstruction.CSupportData;
import mycompiler.mytypereconstruction.CTriple;
import mycompiler.mytypereconstruction.set.CSubstitutionSet;
import mycompiler.mytypereconstruction.set.CTripleSet;
import mycompiler.mytypereconstruction.set.CTypeAssumptionSet;
import mycompiler.mytypereconstruction.typeassumption.CTypeAssumption;
import mycompiler.mytypereconstruction.unify.Unify;
import org.apache.log4j.Logger;

public class ForStmt extends Statement
{
    private Expr head_Initializer_1;
       private Expr head_Condition_1;
       private Expr head_Loop_expr_1;
       private Expr head_Initializer;
       private Expr head_Condition;
```

```
    private Expr head_Loop_expr;
    private Statement body_Loop_block;

    public ForStmt(int offset, int variableLength)
  {
      super(offset,variableLength);
  }

  void sc_check(Vector<Class> classname, Hashtable ch, Hashtable bh, boolean ext,
Hashtable parach, Hashtable parabh)
  throws SCStatementException
  {

  }

  public void codegen(ClassFile classfile, CodeAttribute code, Vector paralist)
  throws JVMCodeException
  {

  }

  /**
   * Implementierung des Algorithmus 5.23 von Martin Plï¿½micke
   * <br/>Achtung Workaround: RefType "Boolean" muss noch durch BaseType
   * "BooleanType" ersetzt werden.
   * <br>Author: Jï¿½rg Bï¿½uerle
   * @param sigma
   * @param V
   * @param supportData
   * @return
   */

  public CTripleSet TRStatement(CSubstitutionSet sigma, CTypeAssumptionSet V,
CSupportData supportData)
  {
    CTripleSet start_set = new CTripleSet();

    CTripleSet init_set = new CTripleSet();

    CTripleSet cond_set = new CTripleSet();

    CTripleSet return_set = new CTripleSet();

    //Expressions kopieren für die Typrekonstruktion um Codegen nicht zu tangieren
```

```
	head_Initializer_1 = head_Initializer;
	head_Condition_1 = head_Condition;
	head_Loop_expr_1 = head_Loop_expr;

	//Fälle überprüfen, in denen nicht alle drei Expressions im Schleifenkopf
existieren

	//Überprüfen, ob eine Expression existiert und diese in ersatz speichern
	boolean condition_exists = true;
	Expr ersatz = null;
	if(head_Initializer_1 == null){
		if(head_Condition_1 == null){
			condition_exists = false;
			if(head_Loop_expr_1 == null){
				//Falls keine Expression existiert, nur TRStatement mit Loop-
Block
				return_set = body_Loop_block.TRStatement(sigma,
V,supportData);
				return return_set;
			}
			else ersatz = head_Loop_expr_1;
		}
		else ersatz = head_Condition_1;
	}
	else ersatz = head_Initializer_1;

	//Falls mindestens eine Expression existiert, werden die leeren mit ersatz
überschrieben
	if(ersatz != null){
		if(head_Initializer_1 == null) head_Initializer_1 = ersatz;
		if(head_Condition_1 == null) head_Condition_1 = ersatz;
		if(head_Loop_expr_1 == null) head_Loop_expr_1 = ersatz;
	}

	//Folgender Code wird ausgeführt, wenn mindestens eine Expression existiert
	start_set = head_Initializer_1.TRExp(sigma,V,supportData);
	Iterator<CTriple> it_init = start_set.getIterator();

	//---------------------------
		// Initializer-Typ rekonstruieren:
		// ---------------------------
	int successfulls1 = 0;
	 Vector<CTypeReconstructionException> exceptions1 = new
Vector<CTypeReconstructionException>();
```

```
      while( it_init.hasNext() ) {
//            CTriple initTriple = it_init.next();
//            init_set.unite(head_Condition_1.TRExp(initTriple.getSubstitutions(),
initTriple.getAssumptionSet(),supportData));
//            //init_set = head_Condition_1.TRExp(initTriple.getSubstitutions(),
initTriple.getAssumptionSet(),supportData);
//            Type r = initTriple.getResultType();

            try{
               CTriple initTriple = it_init.next();
               CTripleSet condit = head_Condition_1.TRExp(initTriple.getSubstitutions(),
initTriple.getAssumptionSet (),supportData);
               init_set.unite( condit );
               //init_set = head_Condition_1.TRExp(initTriple.getSubstitutions(),
initTriple.getAssumptionSet(),supportData);
               Type r = initTriple.getResultType ();
               successfulls1++;
            } catch (CTypeReconstructionException tre) {
               exceptions1.addElement(tre);
            }
      }

      if (successfulls1 == 0) {
            if (exceptions1.size() == 1) {
               throw exceptions1.elementAt(0);
            }
            throw new CTypeReconstructionException(
                  "ForStmt: Es konnte keine Assumption gefunden werden, die auf die
Anforderung passt.",
                  exceptions1, this);
        }

      Iterator<CTriple> it_cond = init_set.getIterator();

      while(it_cond.hasNext()){
            CTriple condTriple = it_cond.next();

            //Wird ausgeführt, falls die Condition existiert
            if(condition_exists == true){
                  // ---------------------------
                  // Rückgabe der Condition mit Boolean unifizieren:
                  // ---------------------------
                  Vector<Vector<Pair>> cond_poss =
Unify.unify(condTriple.getResultType(), new RefType("java.lang.Boolean",getOffset()),
supportData.getFiniteClosure());
                        //
```

```
	System.out.println("cond_set:"+cond_set.toString());
                        //
	System.out.println("condTriple:"+condTriple.toString());
                if (cond_poss.size() != 0) {
                                // ---------------------------
                                // Alle möglichen Unifier anwenden:
                                // ---------------------------

                                int successfulls = 0;
                                Vector<CTypeReconstructionException> exceptions = new
Vector<CTypeReconstructionException>();

                                for (int i = 0; i < cond_poss.size(); i++) {

                                        // ---------------------------
                                        // Condition-Typ rekonstruieren:
                                        // ---------------------------
                                        try {

                                                CSubstitutionSet unifier = new
CSubstitutionSet(cond_poss.elementAt(i));
                                                CTriple boolTriple =
condTriple.cloneAndApplyUnify(unifier);

	cond_set.unite(head_Loop_expr_1.TRExp(boolTriple.getSubstitutions(),
                                                                boolTriple.getAssumptionSet(),
supportData));

                                                successfulls++;
                                        } catch (CTypeReconstructionException tre) {
                                                exceptions.addElement(tre);
                                        }

                                        if (successfulls == 0) {
                                                if (exceptions.size() == 1) {
                                                        throw exceptions.elementAt(0);
                                                }
                                                throw new CTypeReconstructionException(
                                                        "ForStmt: Es konnte keine
Assumption gefunden werden, die auf die Anforderung passt.",
                                                        exceptions, this);
                                        }

                                }
                        } else {
                                throw new
```

```
CTypeReconstructionException("ForStmt.TRStatement(): Bedingung muss boolean sein!",this);
                        }
            }
            //Wird ausgeführt, falls keine Condition existiert
            else{
                  int successfulls2 = 0;
             Vector<CTypeReconstructionException> exceptions2 = new
Vector<CTypeReconstructionException>();
              try{
                  CTripleSet condit =
head_Loop_expr_1.TRExp(condTriple.getSubstitutions(), condTriple.getAssumptionSet
(),supportData);
                  cond_set.unite( condit );
                  successfulls2++;
              } catch (CTypeReconstructionException tre) {
                  exceptions2.addElement(tre);
              }

            if (successfulls2 == 0) {
                  if (exceptions2.size() == 1) {
                      throw exceptions2.elementAt(0);
                  }
                  throw new CTypeReconstructionException(
                          "ForStmt: Es konnte keine Assumption gefunden werden, die auf
die Anforderung passt.",
                          exceptions2, this);
              }
            }
      }

      Iterator<CTriple> it_loop_block = cond_set.getIterator();

      // --------------------------
            // Loop-Expression-Typ rekonstruieren:
            // --------------------------
      while( it_loop_block.hasNext() ) {
            CTriple loop_blockTriple = it_loop_block.next();

      return_set.unite(body_Loop_block.TRStatement(loop_blockTriple.getSubstitutions(),
loop_blockTriple.getAssumptionSet(),supportData));

      }

      return return_set;
```

```
    }

    public String toString()
    {
        return "FOR ";
    }

    public void wandleRefTypeAttributes2GenericAttributes(Vector<Type> paralist,
Vector<GenericTypeVar> genericMethodParameters)
    {
        if(body_Loop_block!=null){
            body_Loop_block.wandleRefTypeAttributes2GenericAttributes(paralist,genericMet
hodParameters);
        }

    }

       public void set_head_Initializer(Expr expr) {
              head_Initializer = expr;

       }

       public void set_head_Condition(Expr expr) {
              head_Condition = expr;

       }

       public void set_head_Loop_expr(Expr expr) {
              head_Loop_expr = expr;

       }

       public void set_body_Loop_block(Statement statement) {
              body_Loop_block = statement;

       }

       public boolean addOffsetsToStatement(CTypeAssumption localAssumption, String
NameVariable, boolean isMemberVariable)
    {

head_Initializer_1.addOffsetsToStatement(localAssumption,NameVariable,isMemberVariable);

body_Loop_block.addOffsetsToStatement(localAssumption,NameVariable,isMemberVariable);
```

```
        return true;
    }
}
```

Abschnitt für die For-Schleife in der Grammatik JavaParser.jay

```
forstatement
                                    //Bsp: for(i=0 ; i<10 ; i++){System.out.println(i)}
                                                    : FOR '(' expression ';'
expression ';' expression ')' statement
                                    {
                                            ForStmt Fst = new
ForStmt($3.getOffset(),$3.getVariableLength());
                                            Fst.set_head_Initializer($3);
                                            Fst.set_head_Condition($5);
                                            Fst.set_head_Loop_expr($7);
                                            Fst.set_body_Loop_block($9);

                                            //Typannahme
                                            $$ = Fst;
                                    }
                                                    //Bsp: for(i=0 ; i<10 ; )
{System.out.println(i)}
                                    |                    FOR '(' expression ';' expression
';' ')' statement
                                    {
                                            ForStmt Fst = new
ForStmt($3.getOffset(),$3.getVariableLength());
                                            Fst.set_head_Initializer($3);
                                            Fst.set_head_Condition($5);
                                            Fst.set_body_Loop_block($8);

                                            //Typannahme
                                            $$ = Fst;
                                    }
                                                    //Bsp: for(i=0 ;  ;
i++){System.out.println(i)}
                                    |                    FOR '(' expression ';' ';'
expression  ')' statement
                                    {
                                            ForStmt Fst = new
ForStmt($3.getOffset(),$3.getVariableLength());
                                            Fst.set_head_Initializer($3);
                                            Fst.set_head_Loop_expr($6);
                                            Fst.set_body_Loop_block($8);

                                            //Typannahme
                                            $$ = Fst;
                                    }
                                                    //Bsp: for( ; i<10 ;
i++){System.out.println(i)}
                                    |                    FOR '(' ';' expression ';'
expression ')' statement
                                    {
                                            ForStmt Fst = new
ForStmt($4.getOffset(),$4.getVariableLength());
                                            Fst.set_head_Condition($4);
                                            Fst.set_head_Loop_expr($6);
                                            Fst.set_body_Loop_block($8);

                                            //Typannahme
                                            $$ = Fst;
                                    }
                                                    //Bsp: for(i=0 ;  ; )
```

```
{System.out.println(i)}
                                    |                    FOR '(' expression ';'  ';'  ')'
statement
                                    {
                                          ForStmt Fst = new
ForStmt($3.getOffset(),$3.getVariableLength());
                                          Fst.set_head_Initializer($3);
                                          Fst.set_body_Loop_block($7);

                                          //Typannahme
                                          $$ = Fst;
                                    }
                                                       //Bsp: for( ; i<10 ; )
{System.out.println(i)}
                                    |                    FOR '(' ';' expression ';'  ')'
statement
                                    {
                                          ForStmt Fst = new
ForStmt($4.getOffset(),$4.getVariableLength());
                                          Fst.set_head_Condition($4);
                                          Fst.set_body_Loop_block($7);

                                          //Typannahme
                                          $$ = Fst;
                                    }
                                                       //Bsp: for( ;  ;
i++){System.out.println(i)}
                                    |                    FOR '(' ';'  ';' expression ')'
statement
                                    {
                                          ForStmt Fst = new
ForStmt($5.getOffset(),$5.getVariableLength());
                                          Fst.set_head_Loop_expr($5);
                                          Fst.set_body_Loop_block($7);

                                          //Typannahme
                                          $$ = Fst;
                                    }
                                                       //Bsp: for( ;   ; )
{System.out.println(i)}
                                    |                    FOR '(' ';' ';' ')' statement
                                    {
                                          ForStmt Fst = new
ForStmt($6.getOffset(),$6.getVariableLength());
                                          Fst.set_body_Loop_block($6);

                                          //Typannahme
                                          $$ = Fst;
                                    }
```