Studienarbeit

Java Typinferenz mit Wildcards

Fachrichtung Informationstechnik

An der Berufsakademie Stuttgart
Außenstelle Horb

Eingereicht von:

Arne Lüdtke
Kirschenrain 16

71126 Gäufelden-Öschelbronn

Ausbildungsbetrieb:

EISENMANN AG
Tübinger Straße 81

71032 Böblingen

Studienjahrgang:	**IT2004**
Matrikelnummer:	**108074**
Bearbeitungszeitraum:	**01.Oktober 2006 – 15. Juni 2007**
Betreuer:	**Prof. Dr. Martin Plümicke**

Dateiname: Studienarbeit.doc / Ersteller: Arne Lüdtke / Datum: 13.07.2007

I. Ehrenwörtliche Erklärung

gemäß § 19 Abs. 4 der Verordnung des Wissenschaftsministeriums über die Ausbildung und Prüfung der Studierenden der Berufsakademien im Ausbildungsbereich Technik vom 27.05.2003. Ich habe die vorliegende Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ort	Datum	Unterschrift

II. Zusammenfassung

In der Programmiersprache Java ist der Programmierer, wie in anderen typisierten Sprachen, gezwungen bei der Deklaration von Methoden und Variablen den Typ anzugeben. Dabei ist in den vorangegangenen Studienarbeiten der Prototyp eines Java Compilers entwickelt worden, welcher mit Hilfe einer Typinferenz die fehlenden Typen berechnet. Dabei basiert die Typinferenz auf dem von Martin Plümicke entwickelten Typunifikations Algorithmus.

Diese Studienarbeit beschäftigt sich mit der Erweiterung des Typunfifkations Algorithmus um die Java Wildcards.

III. Abstract

In the programm language Java the programmer still has to declair the type of each method or variable used in the programm. In past student research projekts the prototype of a Java compiler has been developed, which calculates the types using a typeinference system. The typeinference system is based upon the typeunifikation algorithem developed by Martin Plümicke.

This research project is about the enhancement of the typeunifikation algorithem, so that it can deal with the Java Wildcards.

IV. Vorwort

Ich möchte an dieser Stelle Prof. Dr. Martin Plümicke danken, für die Unterstützung während der Studienarbeit und die Zeit die er sich für Diskussionen über theoretische Grundlagen oder Implementierungsfragen genommen hat. Es machte mir Spaß an dieser Studienarbeit zu arbeiten, da sie eine gute Mischung aus komplexen theoretischen Überlegungen, und deren Implementierung ist.

An dieser Stelle auch Danke an Timo Holzherr, der sich zu beginn des 5. Semesters an einem Samstag die Zeit genommen hat uns alle in das Projekt und die Struktur des Compilers einzuweisen.

V. Inhaltsverzeichnis

VI. Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Seit der Erweiterung des Java Typsystems in Version 5.0 um generische Typen kann der Typ einer Variable beliebig komplex werden. Diese Typangabe, welche nicht nur bei der Variablendeklaration, sondern auch bei jeder Methode, erforderlich ist, kann schnell unübersichtlich und arbeitsaufwändig werden. So gibt es beispielsweise ein Programm, in einer Autohauskette, in welchem zu jedem Auto die Kennzeichen als Index zugewiesen sind. Gleichzeitig sollen die einzelnen Listen über das Autohaus indexiert sein. Eine passende Datenstruktur sähe z.B. so aus:

```
Hashtable<String,Hashtable<String,Auto>> list = new Hashtable<String,Hashtable<String,Auto>>();
```

Dieses Beispiel zeigt, das durch den Einsatz von generischen Typen die Syntax und damit die Deklaration beliebig komplex werden kann. Für diesen Anwendungsfall wurde das Typinferenzsystem entwickelt. Es soll dem Programmierer die Arbeit erleichtern, indem es selbstständig die Typen berechnet. Der Programmierer schreibt das Programm ohne Angabe von den Datentypen.

Eine weitere Aufgabe der Typinferenz ist es bei der Vergabe von Typen die allgemeinste Möglichkeit zu finden. Gerade bei Verwendung von Wildcards in Verbindung mit generischen Typen ist es meistens nicht mehr so einfach als Programmierer den im Sinne der Code-Wiederverwendbarkeit bestmöglichen Typ zu deklarieren.

Das folgende Beispiel stellt eine Klasse dar, welche für eine Matrix steht. In dieser Klasse ist eine Methode mul deklariert, welche die Multiplikation zweier Matrizen durchführt. Die Typen welche vom Programmierer deklariert werden müssen, sind in diesem Beispiel fett gedruckt.

```
class Matrix extends Vector<Vector<Integer>>
{
  Matrix mul(Matrix m)
  {
      Matrix ret = new Matrix (); int i = 0;
      while( i < size() )
      {
        Vector<Integer> v1 = this.elementAt(i);
        Vector<Integer> v2 = new Vector<Integer> ();
        int j = 0;
        while( j < v1.size() )
        {
          int erg = 0; int k = 0;
          while( k < v1.size() )
          {
              erg = erg + v1.elementAt(k).intValue() * m.elementAt(k).elementAt(j).intValue();
              k++;
```

```
            }
            v2.addElement(new Integer(erg));
            j++;
          }
          ret.addElement(v2);
          i++;
        }
        return ret;
    }
}
```

Durch das Typinferenzsystem wird die Angabe der Typen unnötig. Das zweite Beispiel zeigt, wie es dann aussieht.

```
class Matrix extends Vector<Vector<Integer>>
{
   mul(m)
   {
        ret = new Matrix (); i = 0;
        while( i < size() )
        {
          v1 = this.elementAt(i);
          v2 = new Vector<Integer> ();
          j = 0;
          while( j < v1.size() )
          {
            erg = 0; k = 0;
            while( k < v1.size() )
            {
                 erg = erg + v1.elementAt(k).intValue() * m.elementAt(k).elementAt(j).intValue();
                 k++;
            }
            v2.addElement(new Integer(erg));
            j++;
          }
          ret.addElement(v2);
          i++;
        }
        return ret;
   }
}
```

1.2. Aufgabenstellung

Aufgabenstellung dieser Studienarbeit ist die Erweiterung des Typunifikation Algorithmus um die Java Wildcards welche im Java 5.0 Typsystem enthalten sind. Dazu gehört auch die Erweiterung des Typsystems des Compilers um die dafür erforderlichen Klassen sowie die Erweiterung der Grammatik des Parsers, so das dieser deklarierte Wildcards erkennt. Die Implementierung folgender Punkte war Bestandteil dieser Arbeit:

- smaller
  - smaller0
  - smaller1
  - smaller2
  - smaller3
  - smaller4
- greater
  - greater0
  - greater1
  - greater2
  - greater3
- CaptureConversion
- adapt (Modifikationen)
- adaptSup
- adaptExt
- reduce1(Modifikationen)
- reduceSup
- reduceExt
- reduceUp
- reduceLow
- reduceUpLow
- erase1
- erase2
- unifyWC
- match (Modifikationen)
- Diverse Hilfsmethoden (z.B. cartProdukt, DelFreshWildcardTypeVar)

2. Grundlagen

2.1. Compiler

Nach [Bae05] ist ein Compiler ein Programm, welches ein anderes Programm, geschrieben in einer Quellsprache, in eine semantisch Äquivalente andere Sprache übersetzt. Diese Zielsprache ist in den meisten Fällen eine maschinennahe Sprache. Der Compiler soll aber auch Programmierfehler erkennen und melden. Der Übersetzungsprozess kann in drei Schritte aufgeteilt werden:

2.1.1. Scannen / Parsen

Während dem Scannen und Parsen wird die Quellcode Datei eingelesen, und der Text in einem Objektbaum abgebildet. Dies wird durch zwei Programme erledigt. Der Scanner parst die Quelldatei auf Grundlage von Regular Expressions (Chomski 3). Sämtliche Schlüsselwörter und Elemente der Sprache werden erkannt und in Tokens gespeichert. Danach werden die einzelnen Tokens an den Parser weitergereicht. Dieser arbeitet z.B. als LL Parser (Chomski 2), und kann daher die gesamte Sprache abbilden. Hierzu verfügt der Parser über eine Datei in welcher die Grammatik der Sprache enthalten ist. Auf Grundlage dieser Grammatik erkennt der Parser Syntaxfehler im Programm. Der Parser erzeugt für die einzelnen Programmgruppen Objektinstanzen im Speicher, welche er in einem Objektbaum verknüpft. Nach dem Parsen liegt ein vollständiges Abbild der Datei in Form eines abstrakten Syntaxbaumes im Speicher vor. Der Abstrakte Syntaxbaum wird als solcher bezeichnet, da er von der konkreten Sprache abstrahiert.

2.1.2. Semantik Check

Da der Parser nicht den kompletten Objektbaum zur Verfügung hat, können nicht alle Fehler im Quellcode vom Parser erkannt werden. Der Semantik Check läuft durch den kompletten Baum und überprüft diesen auf Fehler. So kann der Parser z.B. nicht erkennen, ob die verwendete Variable i auch wirklich deklariert wurde, oder ob die Typen übereinstimmen. Zudem erkennt der Semantic Check auch weitere Fehler z.B. die mehrfache Verwendung des selben Modifiers (public public void Method...). Eine weitere wichtige Aufgabe ist es, zu den Verweisen auf Variablen, die im ganzen Code verteilt sind den entsprechenden Typ zu ermitteln. Da diese Aufgabe von dem Typrekonstruktionsalgorithmus (TRA) übernommen wird, verzichtet der Prototyp auf einen Semantic Check. Dadurch sind aber andere Syntax Fehler, die nicht vom TRA abgefangen werden möglich.

2.1.3. Die Codegenerierung

Nach dem Semantik Check liegt ein korrektes Abbild des Programms im Speicher. Dieses Abbild ist Plattformunabhängig, es kann in jede beliebige Syntax übersetzt werden. Der Compiler durchläuft in der Generierungsphase das Speicherabbild und schreibt dieses in Form der Zielsprache wieder in eine Datei.

2.2. Bestehendes Programm

Grundlage der Studienarbeit ist ein Java Compiler welcher von den Studenten TIT2000 entwickelt wurde. In vorangegangenen Studienarbeiten wurde dieser Compiler um Generische Datentypen erweitert. 2004 / 2005 wurde dann der Typenrekonstruktionsalgorithmus entwickelt und implementiert.

Diese Typinferierung funktioniert bis auf einige Einschränkungen. Eine dieser Einschränkungen war die Unterstützung von Java Wildcards. Das bestehende Programm setzt sich im wesentlichen aus zwei Teilen zusammen. Teil 1 ist die eigentliche Typinferenz. Der Typenrekonstruktionsalgorithmus, welcher durch Jörg Bäuerle in seiner Studienarbeit 2005 implementiert wurde bildet diesen 1. Teil. Dieser Algorithmus wandert durch den eigentlichen Syntax Baum des Programms, und ermittelt für jede Expression in dem Programm einen Typ. Hierzu erstellt der Typinferenzalgorithmus eine Liste mit diversen Bedingungen die für eine Expression gelten. Diese Liste wird dann an den Typunifikationsalgorithmus übergeben, der den 2. Teil darstellt. Dieser versucht anhand diverser Regeln und Abläufe eine Lösung für das gestellte Problem zu finden. Der *Unify* berechnet dabei alle Lösungen, die für die gegebenen Bedingungen möglich sind. Der Benutzer hat dann die Möglichkeit aus den errechneten Typen auszuwählen, welchen Typ eine Variable annehmen soll. Diese Studienarbeit beschäftigt sich vor allem mit dem 2. Teil. Dieser wird erweitert, sodass er die Wildcards verarbeiten kann. Der Unify wiederum spaltet sich in mehrere Schritte auf (siehe 2.5). Interessant für diese Studienarbeit war vor allem die Methode *sub_unify* welche den 1. Schritt implementiert. In dieser Methode wurden die meisten Änderungen vorgenommen. sub_unify ruft in einer Schleife die einzelnen Regeln auf, bis keine Regel mehr angewendet werden kann. Zu Beginn der Studienarbeit wurde zunächst versucht Ordnung in die Methode zu bringen. So gab es Seitenweise auskommentierten Code, der die Lesbarkeit stark beeinträchtigte. Da das ganze Projekt in einem CVS System eingecheckt ist, wurde dieser auskommentierte Code gelöscht, da er im Zweifelsfall wieder aus dem CVS geholt werden kann. Zusätzlich wurde sub_unify weiter untergliedert, indem größere Regeln in Untermethoden mit entsprechendem Namen ausgelagert wurden.

2.3. Generische Typen

Generische Typen sind vor allem bei Collections bekannt. Vor Java 5.0 gab es natürlich schon Collection Objekte in Java z.B. Vector oder Hashtable. Diese Collections arbeiteten aber immer mit Objekten. Das bedeutet, das man in eine Vectorinstanz andere Objekte beliebigen Typs einfügen konnte. Beim entnehmen der Objekte, musste dann immer ein Typecast durchgeführt werden, was unschön ist, und gegebenenfalls zu einer TypeCastException führt, falls falsche Objekte in der Collection sein sollten. Durch Generische Typen ist es möglich beim Anlegen der Instanz anzugeben, von welchem Typ sie ist. Im Vector fall bedeutet das, dass nur Objekte von diesem Typ hinzugefügt werden können. Entsprechend kommen auch nur Objekte des Typs wieder heraus. Dadurch wird das ganze Programm sicherer, da keine Typecasts mehr notwendig sind, und die meisten Typfehler bereits zur Compilezeit erkannt werden.
Beispiel für eine generische und nicht generische Liste.

```
Vector nonGenericList = new Vector();
nonGenericList.add(new Integer(1)); //Einfügen von Integer
nonGenericList.add(„String“); //Einfügen von String
Object o = nonGenericList.ElementAt(0); //Der Integer wird als Objekt ausgegeben
String i = (String)nonGenericList.ElementAt(1); //Typecast notwendig.

Vector<Integer> genericList = new Vector<Integer>();
genericList.add(new Integer(1)); //Einfügen von Integer
genericList.add(„String“); //Einfügen von String. Compiler wird Fehlermeldung erzeugen.
Integer i = genericList.ElementAt(0); //Entnahme ohne Typecast möglich.
```

2.4. Wildcards

Wenn man sich die Objekte in Java wie einen Vererbungsbaum vorstellt, dann stellt jeder Knoten im Baum genau einen Typ dar. In manchen Fällen wäre es wünschenswert einen Typ zu deklarieren, der als Platzhalter für jeden beliebigen oder für einen eingegrenzten Typbereich steht. Diese Platzhalter werden im Java Typsystem Wildcards genannt. Gerade, um Methoden zu schreiben, die wieder verwendet werden können, sind Wildcards ein wichtiges Werkzeug. Wildcards können nur anstelle von Typparametern deklariert werden, eine direkte Deklaration ist nicht möglich. Beispiel:

```
Vector<? extends Integer> vec = new Vector<? extends Integer>(); // Extends Wildcard als Parameter.
? extends Integer wcType = new ? extends Integer(); //Direkte Deklaration ist nicht möglich.
```

Es gibt drei verschiedene Wildcards. Die Allgemeine Wildcard, welche für einen beliebigen Typen steht. Die Extends Wildcard, welche bildlich im Objektbaum einen Zweig nach oben begrenzt. Das bedeutet, alle Klassen müssen in der Vererbung kleiner oder gleich sein wie die Wildcard. Als drittes die Super Wildcard. Diese begrenzt den Objektbaum nach Unten. Ergo müssen alle Klassen in der Vererbungshierarchie größer oder gleich sein wie die Wildcard. Anschaulich wird das, wenn man sich das mit dem Collection Typ Vector der Java API anschaut. Generell gilt, dass Vector immer Objekte der oberen Schranke zurückgibt, und Objekte der Unteren Schranke eingefügt werden können.

```
Vector<?> vec1 = ...; //Vector wird irgendwie initialisiert.
Object o = vec1.ElementAt(..); //ElementAt liefert immer ein Object, da alle von Object erben.
vec1.add(..); //Das Einfügen wird verboten, da nicht bekannt ist, von welchem Typ der Vector ist.

Vector<? extends Integer> vec2 = ...;
Integer i = vec2.ElementAt(..); //liefert Integer, da Integer die Obere Schranke ist.
vec2.add(..); //Das Einfügen ist verboten, da die Untere Schranke undefiniert ist.

Vector<? super Integer> vec3 = ...;
Object o = vec3.ElementAt(..); //liefert Object, da Object die Obere Schranke ist.
vec3.add(3); //Es können Integer eingefügt werden, da Integer die untere Schranke ist.
```

Im Folgenden werden die einzelnen Wildcards näher beschrieben, und kleine Codebeispiele gegeben, wie man die einzelnen Wildcards verwenden kann, und weshalb diese sinnvoll sind. Die Beispiele für die allgemeine Wildcard und die Super Wildcard wurden leicht abgeändert aus [Loc06] übernommen.

2.4.1. Die allgemeine Wildcard (?)

Die allgemeine Wildcard steht als Platzhalter für eine beliebige Klasse im Objektbaum. Sie unterscheidet sich von der Klasse Object dadurch, dass Object eine konkrete Klasse ist, von der auch Instanzen angelegt werden können.

```
Object o = new Object(); //Gültiger Java Aufruf;
? o = new ?(); //Fehler.
```

Angenommen, es soll eine Methode geschrieben werden, die basierend auf der ToString() Methode alle Elemente eines Vectors ausdruckt. Dann wäre folgende Methode der erste Ansatz:

```
public void printVector(Vector<Object> c)
{
  for(Object e : c)
  {
    System.out.println(e.ToString());
  }
}

public static void Main(string[] args)
{
  Vector<Object> vecObj = new Vector<Object>();
  //vecObj mit beliebigen Werten füllen.
  printVector(vecObj); //Ok.

  Vector<Integer> vecInt = new Vector<Integer>();
  //vecInt mit beliebigen Werten füllen.
  printVector(vecInt) //Compiler meldet Fehler, da Vector<Integer> ungleich Vector<Object>
}
```

Wie das kleine Beispiel zeigt, kann diese Methode nicht für verschiedene Typen verwendet werden. An dieser Stelle ist eine Wildcard erforderlich. Der Parameter im Übergabevector wird durch die Wildcard ersetzt. Daraufhin funktioniert das Programm.

```
public void printVector(Vector<?> c)
{
  for(Object e : c)
  {
    System.out.println(e.ToString());
  }
}
//Rest unverändert, siehe oben.
```

2.4.2. Die extends Wildcard (? extends Typ)

Angenommen, es gibt ein Java Programm für ein Autohaus. Im Objektbaum gibt es die abstrakte Oberklasse *Auto*. Von dieser Klasse erben die Klassen *BMW, Ford, Fiat*. In der Oberklasse Auto ist die abstrakte Methode *printInfo()* deklariert, die von allen Unterklassen überschrieben wird. Im Programm gibt es Listen der einzelnen Hersteller. Eine Methode soll den Inhalt der Liste drucken.

```
public void printList(Vector<Auto> list)
{
  for (Auto e : list)
  {
    e.printInfo();
  }
}
```

Der Compiler verbietet den Aufruf der Methode wie in folgendem Beispiel:

```
Vector<BMW> vecBMW = new Vector<BMW>();
//vecBMW mit Werten füllen
printList(vecBMW); //Fehler.
```

Eigentlich wäre es wünschenswert, das vecBMW übergeben werden kann, da BMW ja von Auto erbt. Aber folgendes Beispiel wäre dann auch möglich, und verdeutlich, warum solche Aufrufe verboten werden müssen.

```
public void fillVector(Vector<Auto> vec)
{
  vec.add(new BMW());
  vec.add(new Ford());
  vec.add(new Fiat());
}
```

Diese Befehle sind alle Korrekt. In einen Vector<Auto> können alle Objekte eingefügt werden, die von Auto erben. Nun nehmen wir an, dass die Methode fillVector wie zuvor mit einem Vector<BMW> aufgerufen werden könnte. In diesem Fall würde man Ford und Fiat Objekte in einen Vector<BMW> einfügen, was das ganze Typsystem unschlüssig (unsound) macht. Trotzdem möchte man eine Methode, welche den Vector akzeptiert, unabhängig vom Autohersteller. Für diesen Anwendungsfall eignet sich die Extends Wildcard. Die printList Methode wird folgendermaßen abgeändert.

```
public void printList(Vector<? extends Auto> list)
{...}
```

In diesem Fall kann jeder Vector übergeben werden, dessen Typparameter von Auto erbt.

2.4.3. Die super Wildcard (? super Typ)

Angenommen, es existiert eine generische Klasse, welche über einen Comparer mit anderen Klassen verglichen werden kann. Diesen Comparer bekommt sie bei der Instanzierung im Construktor übergeben.

```
class GenClass<E>
{
  private Comparer<E> comp;
  GenClass(Comparer<E> c){
    this.comp = c;
  }

  public static void Main(string[] args)
  {
    Comparer<String> compString = new Comparer<String>();
    Comparer<Object> compObject = new Comparer<Object>();
    GenClass<String> genC_1 = new GenClass<String>(compString); //Funktioniert
    GenClass<String> genC_2 = new GenClass<String>(compObject); //Funktioniert NICHT
  }
}
```

Dadurch, dass die GenClass vom Typ String ist, erwartet der Compiler zwingend einen Comparer vom Typ String. Sinnvoll wäre es aber, wenn ebenfalls der Comparer vom Typ Object möglich wäre. Hierzu wird die Deklaration der Klasse GenClass erweitert.

```
class GenClass<E>
{
  private Comparer<? super E> comp;
  GenClass(Comparer<? super E> c){
    this.comp = c;
  }

  public static void Main(string[] args)
  {
    Comparer<String> compString = new Comparer<String>();
    Comparer<Object> compObject = new Comparer<Object>();
    GenClass<String> genC_1 = new GenClass<String>(compString); //Funktioniert
    GenClass<String> genC_2 = new GenClass<String>(compObject); //Funktioniert jetzt auch.
  }
}
```

Durch die Angabe der Super Wildcard im Constructor von GenClass wird nun jeder Comparer akzeptiert, welcher in der Objekthierarchie über E steht.

2.5. Die Typ Unifikation

Die Unifikation von Typen bedeutet, einen Unifier zu finden, der allen Typvariablen einen Typ zuordnet. Man kann das in etwa mit einem Gleichungssystem vergleichen, dass gelöst werden soll. Als Eingabemenge gibt es diverse Typen, die Unifiziert werden sollen. In diesen Typen sind Variablen enthalten. Die Unifikation soll nun ermitteln, wie diese Variablen ersetzt werden müssen, damit das System stimmig ist. Wie bei Gleichungssystemen können dabei eine, keine und mehrere Lösungen entstehen. Die Unifikation darf nicht mit der Typinferenz verwechselt werden. Die Typinferenz ist ein System, welches in einer Programmiersprache automatisch den Typ der einzelnen Elemente ermittelt. Die Typunifikation ist ein Teil der Inferenz. Der in diesem Projekt verwendete Unifikationsalgorithmus stammt von Martin Plümicke, und basiert auf den Arbeiten von Herbrand, Martelli und Montanari. Die Erweiterung des Algorithmus um Wildcards ist eine Entwicklung von Martin Plümicke.

Der Input des Algorithmus ist eine Menge von Paaren. Ein paar ist dabei eine Collection von genau zwei Typen, welche in einem gewissen Verhältnis zueinander stehen. Dieses Verhältnis entscheidet, ob die beiden Typen nur voneinander erben, oder ob diese Identisch sein sollen. Als Ausgabe liefert der Algorithmus eine Menge von Paaren in der so genanten *Solved Form*. Das bedeutet, die Paare bestehen aus einer Typvariable und einem Typ, und das Verhältnis ist, dass diese gleich sind.

Die Funktionsweise des Algorithmus ist im wesentlichen, immer wieder durch die Eingabemenge zu laufen, und verschiedene Regeln (siehe 2.5.1) auf diese Anzuwenden. Dies macht der Algorithmus solange, bis keine Änderung mehr vorkommt. Die sieben Schritte des Algorithmus sind im folgenden kurz beschrieben. Eine formale Definition ist in [Plu07] zu finden.

1. **Schritt:**

Im ersten Schritt werden die Regeln auf die Eingabemenge angewendet, solange, bis keine Regel mehr angewendet werden kann. Die Ergebnismenge ist Eq

2. **Schritt:**

Im zweiten Schritt werden alle Paare aus der Menge Eq aussortiert, die entweder zwei TypePlaceholder enthalten, oder die keine TypePlaceholder enthalten. Diese Paare werden in die Menge Eq1 eingefügt.

3. **Schritt:**

In diesem Schritt wird die Menge Eq2 erzeugt. Diese Menge ist die Differenz von Eq zu Eq1. Also quasi alle Elemente aus Eq, mit Ausnahme der Elemente, welche in Eq1 sind.

4. **Schritt:**

In diesem Schritt wird die Menge Eq2 durchlaufen, und für jedes Paar wird eine Menge von Ergebnissen gebildet. Über diese Ergebnisse wird das Kreuzprodukt gebildet.

5. **Schritt:**

Mit der *Subst* Regel werden alle Typvariablen, in den Ergebnismengen aus Schritt 4 ersetzt.

6. **Schritt:**

In diesem Schritt werden alle Ergebnissätze geprüft. Für Sätze, die in Regel 5 geändert wurden, wird neu mit dem 1. Schritt begonnen. Dies geschieht durch den rekursiven Aufruf von unifyWC. Für die anderen wird eine Vereinigung mit einer Ergebnismenge durchgeführt.

7.Schritt:

Aus den berechneten Sets werden alle aussortiert, die in *Solved Form* sind. Solved Form ist, wenn die Ergebnismenge nur noch aus Paaren besteht, deren linker Teil ein TypePlaceHolder, und der rechte Teil ein entsprechender Typ ist.

Im Rahmen dieser Studienarbeit wurden die ursprünglichen fünf Regeln aus sub_unify, welche in [Ott04] implementiert und in darauf folgenden Studienarbeiten erweitert wurden, zum Teil abgeändert. Zusätzlich wurden weitere zehn Regeln implementiert. Folglich existieren nun 15 Regeln. Für die Regeln, welche die Wildcards unifizieren ist zusätzlich die Funktion *greater* erforderlich. Diese Funktion ermittelt alle Typen welche in der Hierarchie größer sind, also Superklassen des Eingabetyps darstellen. *smaller* ermittelt demnach das Gegenteil von *greater* also alle Typen die kleiner sind. smaller und greater rufen sich wechselseitig rekursiv auf. Die *CaptureConversion* (kurz CC) wird in diversen Regeln sowie in *greater* und *smaller* verwendet.

2.5.1. Die zehn neuen Regeln

Die Schreibweise der Regeln: $\frac{Vorraussetzung1, Vorraussetzung2, \ldots}{Ergebnis} Annahme$

Die Regeln ermöglichen es, aus bestehenden Informationen, auf neue zu schließen. Dadurch erst wird die Unifikation möglich. Deshalb sind die Regeln der wesentliche Bestandteil des Algorithmus.

Die erase Regeln:

Im alten Algorithmus gab es nur eine Regel, welche für das Löschen von Paaren aus der Ergebnismenge zuständig ist. Diese Regel wurde in *erase3* umbenannt. Zusätzlich wurden zwei weitere Regeln implementiert. Das frühzeitige Löschen von Paaren aus der Menge beschleunigt den Algorithmus, da keine aufwändigeren Operationen mehr mit diesen Paaren ausgeführt werden.

$$(\text{erase}1) \quad \frac{Eq \cup \{\theta \preccurlyeq \theta'\}}{Eq} \quad \theta \leq^* \theta'$$

$$(\text{erase}2) \quad \frac{Eq \cup \{\theta \preccurlyeq_? \theta'\}}{Eq} \quad \theta' \in \mathbf{grArg}(\theta)$$

$$(\text{erase}3) \quad \frac{Eq \cup \{\theta \doteq \theta'\}}{Eq} \quad \theta = \theta'$$

Die adapt Regeln:

Mit der Implementierung der Wildcard Unifizierung wurden zwei zusätzliche adapt Regeln notwendig, welche auf den Wildcards operieren. Diese adapt Regeln arbeiten im wesentlichen Äquivalent zu der bisherigen Regel. Aufgabe ist es, über die Finite Closure (FC), in welcher die Vererbungshierarchien gespeichert sind, neue Typen zu ermitteln. Beispiel: {Stack<A> < Vector<Integer>} -> {Vector<A> < Vector<Integer>}, da {Stack<a> < Vector<a>} in der FC steht.

Die reduce Regeln:

Reduce Regeln werden verwendet, um die Typparameter aus Typen herauszureduzieren.
Beispiel: {Vector<A> < Vector<Integer>} -> {A < Integer}.

Im alten Algorithmus gab es zwei Regeln, *reduce1* und *reduce2*. Diese Regeln wurden mit kleinen Änderungen in *reduce1* übernommen. Zusätzlich wurden sechs weitere Reduce Regeln implementiert. Die Regeln *reduceUp, reduceLow* und *reduceUpLow* dienen dazu aus Wildcard Typen die Wildcards zu entfernen.
Beispiel: Aus {A < ? super B} -> {A < B}. Oder {? extends A < B} -> {A < B}.

Die Regeln *reduceExt* und *reduceSup* arbeiten auf Wildcard Typen, wie die *reduce1* Regel auf normalen Typen.

$$\text{(reduceUp)} \quad \frac{Eq \cup \{\, \theta \ll {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}} \qquad \text{(reduceUpLow)} \quad \frac{Eq \cup \{\, {}_{?}\theta \ll {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}}$$

$$\text{(reduceLow)} \quad \frac{Eq \cup \{\, {}_{?}\theta \ll \theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}}$$

$$\text{(reduce1)} \quad \frac{Eq \cup \{\, C\langle\theta_1, \ldots, \theta_n\rangle \ll D\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_{\pi(1)} \ll_? \theta'_1, \ldots, \theta_{\pi(n)} \ll_? \theta'_n \,\}}$$

where
- $C\langle a_1, \ldots, a_n\rangle \leq^{*} D\langle a_{\pi(1)}, \ldots, a_{\pi(n)}\rangle$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation

$$\text{(reduceExt)} \quad \frac{Eq \cup \{\, X\langle\theta_1, \ldots, \theta_n\rangle \ll_? {}_{?}Y\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_{\pi(1)} \ll_? \theta'_1, \ldots, \theta_{\pi(n)} \ll_? \theta'_n \,\}}$$

where
- ${}_{?}Y\langle a_{\pi(1)}, \ldots, a_{\pi(n)}\rangle \in \mathbf{grArg}(\, X\langle a_1, \ldots, a_n\rangle \,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation

$$\text{(reduceSup)} \quad \frac{Eq \cup \{\, X\langle\theta_1, \ldots, \theta_n\rangle \ll_? {}^{?}Y\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta'_1 \ll_? \theta_{\pi(1)}, \ldots, \theta'_n \ll_? \theta_{\pi(n)} \,\}}$$

where
- ${}^{?}Y\langle a_{\pi(1)}, \ldots, a_{\pi(n)}\rangle \in \mathbf{grArg}(\, X\langle a_1, \ldots, a_n\rangle \,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation

$$\text{(reduceEq)} \quad \frac{Eq \cup \{\, X\langle\theta_1, \ldots, \theta_n\rangle \ll_? X\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_{\pi(1)} \doteq \theta'_1, \ldots, \theta_{\pi(n)} \doteq \theta'_n \,\}}$$

$$\text{(reduce2)} \quad \frac{Eq \cup \{\, C\langle\theta_1, \ldots, \theta_n\rangle \doteq C\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

2.5.2. Die Funktion *greater*

Die Funktion *greater* liefert die Menge aller Typen welche in der Vererbung größer sind, als der übergebene Typ. Die Funktion arbeitet in drei Unterfunktionen, die jeweils einen Teil der Ergebnismenge erstellen. Die Vereinigung über diese drei Mengen ist das eigentliche Ergebnis. greater1 wendet die Funktion *greater* rekursiv auf die Parameter. Anschließend werden alle Permutationen mit den Ergebnismengen der Parameter erzeugt. greater2 ermittelt über die FC alle größeren Typen. greater3 wendet greater1 auf alle Typen in greater2 an. [Plu07] spezifiziert und beschriebt den Ablauf des Algorithmus detailliert. Für das folgende Beispiel seien zwei Klassen gegeben. Die Klasse *Super<A>* und die Klasse *Sub<A>* welche von *Super<A>* erbt. Des Weiteren die Klassen *ClassA* mit der Subklasse *ClassB*.

Beispiel: greater(Sup<? super ClassB>) geschrieben: greater(Sup<?ClassB>)

greater1(Sup<?ClassB>)
grArg(?ClassB)
smaller(ClassB) = {ClassB}
Erg: grArg(?ClassB): {?ClassB, ClassB}
Erg: greater1(Sup<?ClassB>): {Sup<?ClassB>, Sup<ClassB>}

greater2(Sup<?ClassB>)
Erg: greater2(Sup<?ClassB>): {Super<ClassB>, Super<?ClassB>}

greater3(Sup<?ClassB>)
greater1(Super<ClassB>)
grArg(ClassB)
greater(ClassB) = {ClassA}
smaller(ClassB) = {}
Erg: grArg(ClassB): {ClassB, ?ClassA}
Erg: greater1(Super<ClassB>): {Super<ClassB>, Super<?ClassA>}
greater1(Super<?ClassB>)
grArg(?ClassB)
smaller(ClassB) = {}
Erg: grArg(?ClassB): {}
Erg: greater1(Super<?ClassB>): {}
Erg: greater3(Sup<?ClassB>): {Super<ClassB>, Super<?ClassA>}

Erg: greater(Sup<?ClassB>):
{Sup<?ClassB>, Sup<ClassB>, Super<ClassB>, Super<?ClassB>, Super<?ClassA>}

2.5.3. Die Funktion *smaller*

Die Funktion *smaller* wird verwendet um für einen gegebenen Typ die Menge aller Typen zu finden, welche in der Vererbungshierarchie kleiner sind, also von dem Ausgangstyp erben. Die Funktion arbeitet dabei in vier Teilschritten (smaller1 – smaller4). Diese Unterfunktionen erzeugen jeweils eine Teilmenge von smaller. Am Ende wird der Durchschnitt über diese Teilmengen gebildet. Dieser Durchschnitt ist das Ergebnis. smaller1 ermittelt die Menge aller Typen durch rekursiven Aufruf der Funktion smaller auf den Parametern, und anschließende Permutation dieser. smaller2 bildet die Capture Conversion über die Ergebnismenge von smaller1. smaller3 ermittelt über die FC alle Typen die kleiner sind. smaller4 wandelt die FreshTypeVars die in smaller2 erzeugt wurden zurück. Der genaue Ablauf des Algorithmus ist in [Plu07] genau beschrieben.
Für das folgende Beispiel seien zwei Klassen gegeben. Die Klasse *Super<A>* und die Klasse *Sub<A>* welche von *Super<A>* erbt. Des Weiteren die Klassen *ClassA* mit der Subklasse *ClassB*.

Beispiel: smaller(Super<? extends ClassA>) geschrieben: smaller(Super<?ClassA>)

smaller1(Super<?ClassA>)
smArg(?ClassA)
smaller(ClassA) = {ClassA, ClassB}
Erg: smArg(?ClassA): {?ClassA, ClassA, ?ClassB, ClassB}
Erg: smaller1(Super<?ClassA>):
{Super<?ClassA>, Super<ClassA>, Super<?ClassB>, Super<ClassB>}

smaller2(Super<?ClassA>)
Erg: smaller2(Super<?ClassA>): {Super<X|^{ClassA}>, Super<Y|^{ClassB}>}

smaller3(Super<?ClassA>)
Erg: smaller3(Super<?ClassA>):
{Sup<?ClassA>, Sup<ClassA>, Sup<?ClassB>, Sup<ClassB>, Sup<X|^{ClassA}>, Sup<Y|^{ClassB}>}

smaller4(Super<?ClassA>)
Erg: smaller4(Super<?ClassA>):
{Super<ClassA>, Super<ClassB>, Sup<ClassA>, Sup<ClassB>}

Erg: smaller(Super<?ClassA>)
{Super<?ClassA>, Super<ClassA>, Super<?ClassB>, Super<ClassB>, Sup<?ClassA>, Sup<ClassA>, Sup<?ClassB>, Sup<ClassB>}

2.5.4. Die Funktion *Capture Conversion*

Die CaptureConversion (CC) wandelt eine Wildcard in eine FreshWildcard um. FreshWildcardTypes sind implizite Typen, die nur wärend der Typunifizierung auftreten. Sie können nicht vom Programmierer deklariert werden, da es sie im endgültigen Programm nicht gibt. Da bei der Typunifizierung auch RefTypes entstehen die als Parameter FreshWildcardTypes besitzen, müssen diese Typen über die Funktion *DelFreshWildcardTypeVar* herausgefiltert werden. Die impliziten Typen sind typtheoretisch notwendig, da an manchen Stellen keine Wildcard mehr stehen darf. Durch die Umwandlung wird verhindert, das in den Unify-Rekursionen Regeln angewendet werden, die für Wildcards bestimmt sind. Die CaptureConversion wird in den Regeln adapt, adaptExt, adaptSup, smaller und greater verwendet.

Beispiel: *CaptureConversion(Super<? extends ClassA>) geschrieben: CC(Super<?ClassA>)*
CC(Super<?ClassA>) = {Super<X|^{ClassA}>}

Beispiel: *CaptureConversion(Sup<? super ClassA>) geschrieben: CC(Sup<$^{?}$ClassA>)*
CC(Sup<$^{?}$ClassA>) = {Sup<$_{ClassA}$|Y>}

Eine FreshWildcard besitzt also Upper Bounds (obere Grenzen) wie eine Extends Wildcard, sowie Lower Bounds (untere Grenzen) wie eine Super Wildcard. Nun kann es vorkommen, dass eine Wildcard in einen Generische Typ Variable eingesetzt wird, die selber einen Bound hat. Beispiel:

class Test<a extends Number>: Der Parameter a darf nur durch Typen ersetzt werden, die von Number erben.

CC(Test<? Super Integer>) = {Test<$_{Integer}$|Z|^{Number}>}
CC(Test<? extends Integer> = {Test<Z|$^{Integer\ \&\ Number}$>}

Bei Typen mit mehreren Bounds kann es zu Konflikten kommen, wenn der Untere Bound größer ist als einer der Oberen Bounds. Beispiel:

class Test2<a extends Vector & Stack>: Der Parameter a darf nur durch Typen ersetzt werden, die von Vector und Stack erben. Diese Deklaration ist natürlich nicht sinnvoll, da Stack von Vector erbt, sie hilft aber das Problem zu verstehen.

CC(Test2<? Super Vector>) = {Test2<$_{Vector}$|X|$^{Vector\ \&\ Stack}$>}
Beim Vergleich wird deutlich, dass es keinen Typ geben kann, welcher der Bedingung entspricht. Der Typ soll größer gleich Vector sein, und gleichzeitig kleiner als Vector sein.

3. Erweiterung der Typ Unifikation.

3.1. Erarbeiten der theoretischen Grundlagen

Zu Beginn der Studienarbeit war die Erweiterung des Algorithmus durch Prof. Dr. Martin Plümicke noch nicht fertig gestellt. Das Hauptproblem stellte dabei die Tatsache dar, dass für die Inferierung der Wildcards eine Menge aller Typen erstellt werden muss die in der Vererbung kleiner sind als die obere Schranke der Wildcard. Diese Menge *smaller* wird durch Rekursion unendlich groß.
Beispiel: *smaller(Vector<A>) -> {Vector<A>, Vector<Vector<A>>, Vector<Vector<Vector<A>>>,...)*

Das ist auch der Grund, warum manche Informariker /-innen der Meinung sind, dass die Typ Inferierung von Wildcards in die Klasse der nicht berechenbaren Algorithmen fällt. Dabei arbeitet der gefundene Algorithmus von Martin Plümicke auf der Grundlage, dass ab einem gewissen Rekursionsgrad, alle weiteren Typen in der Ergebnismenge von *smaller* aus den bereits in der Menge enthaltenen zusammengesetzt werden können. Durch diese Annahme, kann die Rekursion in *smaller* abgebrochen werden, und die Menge wird dadurch endlich, und damit berechenbar. Sollte allerdings durch diese Annahme eine unbewusste implizite Einschränkung des Typsystems entstanden sein, arbeitet der Algorithmus natürlich nicht 100%ig korrekt.

Ein wesentlicher Teil dieser Studienarbeit war es, zusammen mit Prof. Dr. Plümicke die theoretischen Grundlagen durchzuarbeiten und eventuelle Fehler zu finden. Erst im Laufe des 5. Semesters entwickelten sich die theoretischen Grundlagen soweit, dass eine Implementierung möglich war. Das Ergebnis zum Ende des 5. Semesters war aber ein Algorithmus, welcher mit vielen Ausnahmen arbeitete. Während der Praxisphase des 5. Semesters wurde der Algorithmus durch Martin Plümicke noch einmal komplett überarbeitet, sodass Mitte Februar der theoretische Teil soweit fortgeschritten war, dass mit der Implementierung begonnen werden konnte.

3.2. Neuimplementierungen und Änderungen

Der bestehende Algorithmus ist in der Klasse Unify implementiert. Zu beginn der Studienarbeit wurden die neuen Funktionen *greater, smaller* und die *CaptureConversion* implementiert. Anschließend wurden die neuen Erase Regeln implementiert. Dazu musste in der Klasse Pair das Boolean Flag bEqual ersetzt werden. Das Flag im Pair signalisierte ob der Operator ein equal, oder ein smaller Operator ist. Da mit der Implementierung der Wildcards drei Operatoren (smaller, smaller extends und equal) notwendig sind, war das Boolean nicht ausreichend. Es wurde durch eine Enumeration ersetzt, welche die drei Zustände annehmen kann. Nach den Erase Regeln wurden die neuen Reduce Regeln implementiert, und anschließend die Adapt Regeln. Zusätzlich musste der Ablauf, der sieben Schritte des Algorithmus neu geschrieben werden.

3.2.1. Implementierung von *greater*

Die Funktion greater spaltet sich in verschiedene Unterfunktionen auf. Die einzelnen Funktionen sollen hier durch Aktivitätsdiagramme veranschaulicht werden.

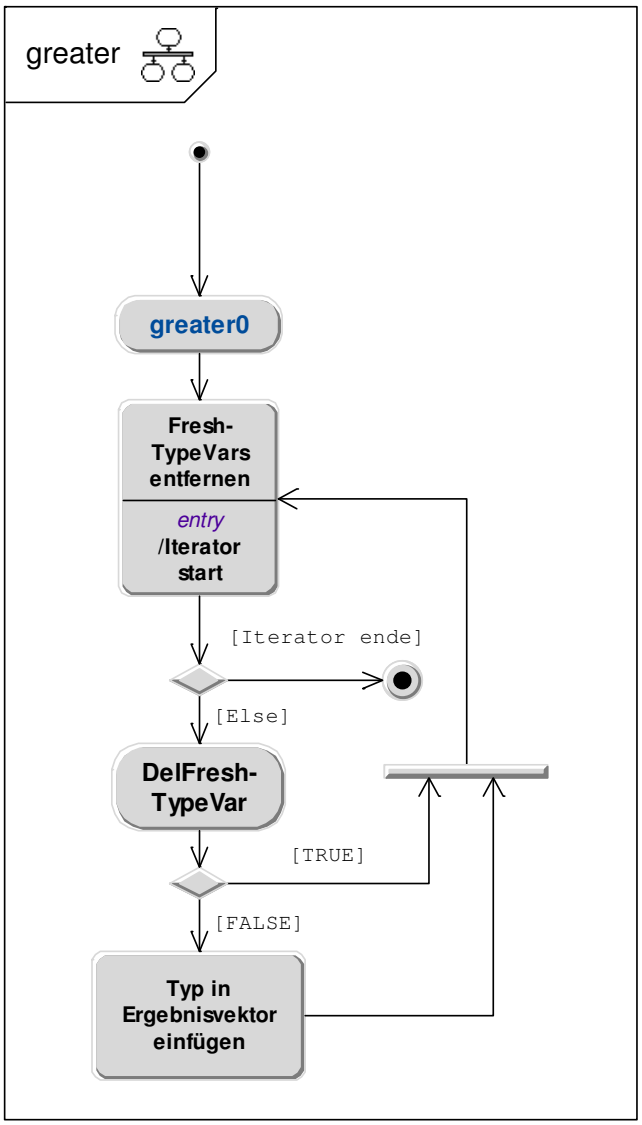


Abbildung 1 - Diagramm greater

greater:

greater dient als Einstieg, wenn die Menge greater Ermittelt werden soll. Da sich greater und smaller wechselseitig rekursiv aufrufen, aber nur am Ende die FreshWildcardTypes entfernt werden, dient für den rekursiven Aufruf greater0 als Einstiegspunkt.

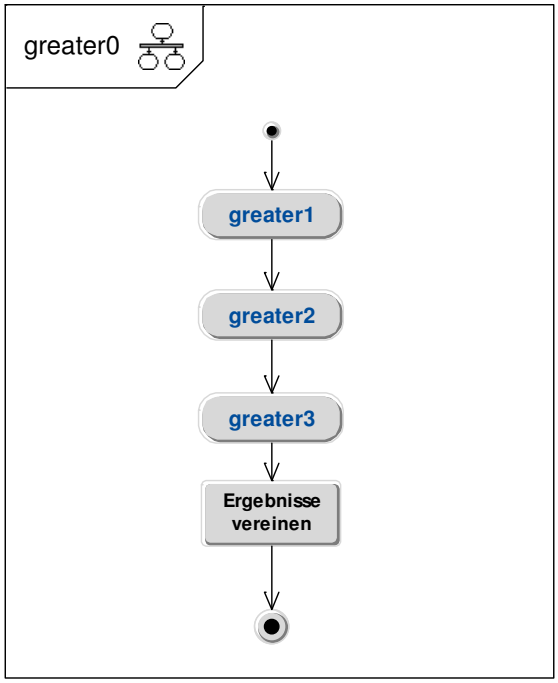


Abbildung 2 - Diagramm greater0

greater0:

greater0 ruft nacheinander die einzelnen Unterfunktionen greater1, greater2 und greater3 auf. Da greater2 auf den Ergebnissen von greater1 arbeitet, und greater3 wiederum auf denen von greater2, reicht greater0 die Ergebnisse durch. greater0 sammelt alle Ergebnisse der Unterfunktionen in einem Ergebnisvektor, welcher von der Funktion zurückgegeben wird.

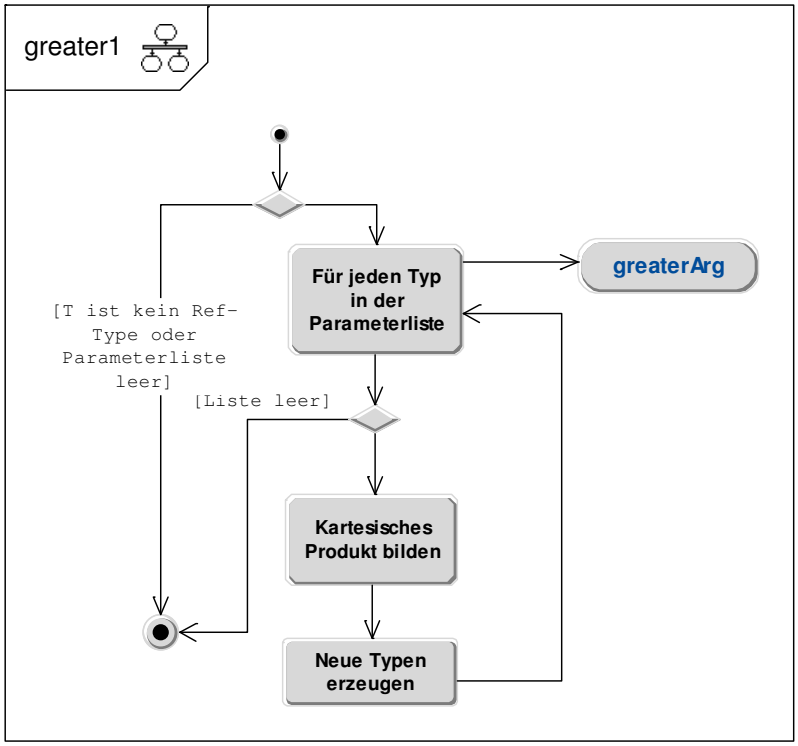


Abbildung 3 - Diagramm greater1

greater1:

greater1 Soll für jeden Parameter des Typs die größeren Typen ermitteln und vom original Typ entsprechend viele Duplikate mit den neuen Parameterlisten anlegen. Sollte es sich bei dem Übergabetyp um keinen RefType handeln, oder sollte die Parameterliste des RefTypes leer sein, dann ist die Ergebnismenge von greater1 leer.

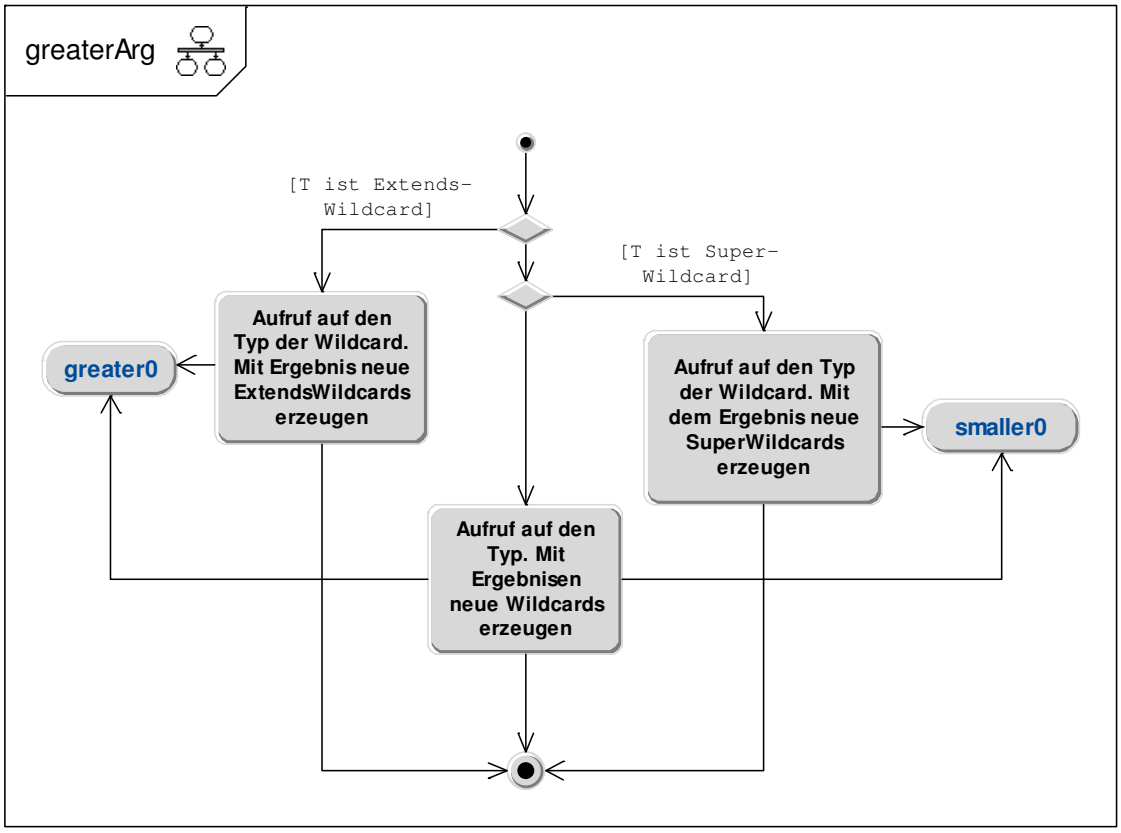


Abbildung 4 – Diagramm greaterArg

greaterArg:

In greaterArg findet der eigentlich rekursive Aufruf statt. Es wird abgefragt, ob der Typ ein Wildcard Type ist. In diesem Fall wird entweder smaller0 oder greater0 aufgerufen. Ist es kein Wildcard Type, dann wird smaller0 und greater0 aufgerufen.

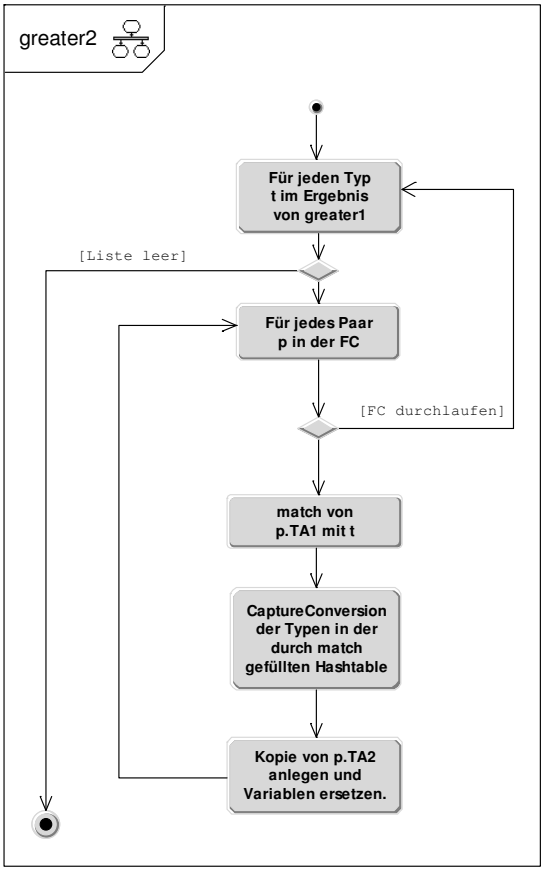


Abbildung 5 – Diagramm greater2

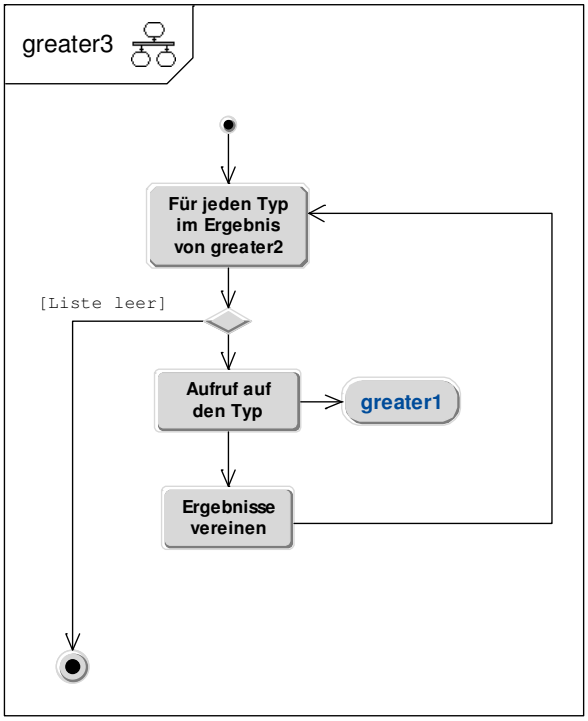


Abbildung 6 – Diagramm greater3

greater2:

greater2 ermittelt auf Basis der Ergebnisse von greater1 und der FC die Menge der Typen die durch Vererbung größer sind.

greater3:

greater3 wendet greater1 auf die Ergebnisse von greater2 an. Dadurch werden die Parametervariationen von greater2 ebenfalls ermittelt.

3.2.2. Implementierung von *smaller*

smaller ist im Ablauf fast identisch zu greater. Auch hier sollen Aktivitätsdiagramme der Veranschaulichung dienen.

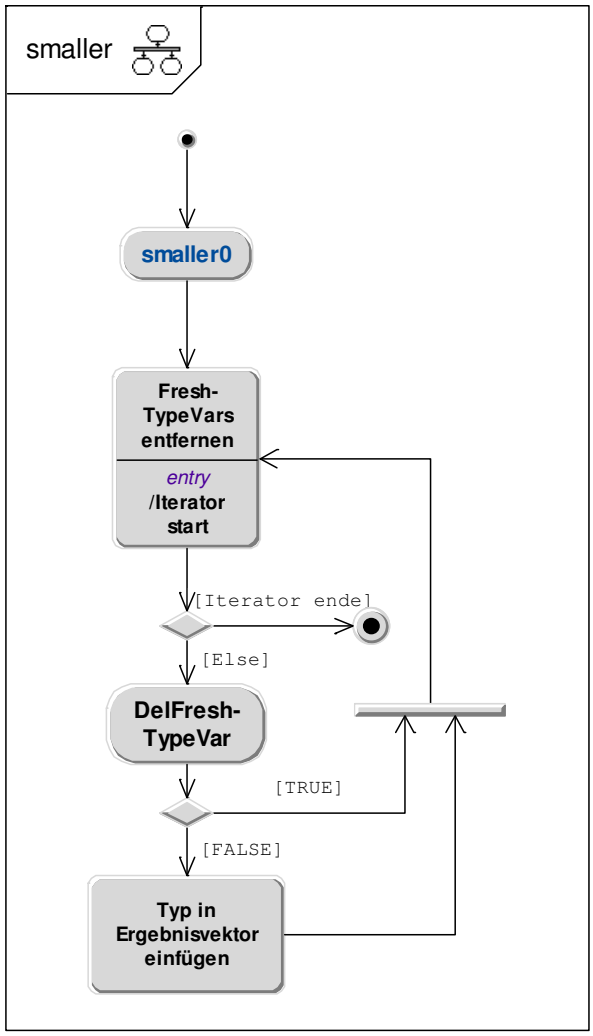


Abbildung 7 – Diagramm smaller

smaller:

Wie bei greater dient smaller als Einstiegspunkt. Es ruft smaller0 auf, und entfernt am Ende alle FreshWildcardTypes. smaller wird in den Regel nicht verwendet, aber bei Erzeugung der Ergebnismengen im 4. Schritt des Unify wird auf smaller verwiesen.

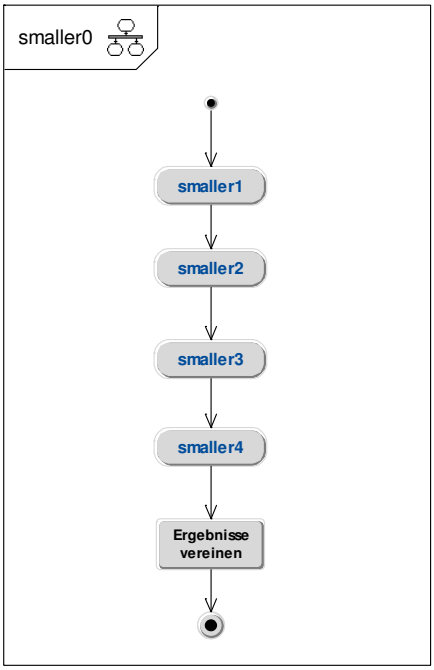


Abbildung 8 – Diagramm smaller0

smaller0:

smaller0 ruft die Unterfunktionen smaller1 – smaller4 auf. Jede Funktion arbeitet auf den Ergebnissen der vorhergehenden Funktion. smaller0 hat die Aufgabe, die Ergebnisse der einzelnen Funktionen zu sammeln und an die darauf folgende Funktion zu übergeben. Am Ende werden die Ergebnisse der Funktionen vereint.

smaller1:

smaller1 arbeitet auf RefTypes und ruft für jeden Parameter des Typs die Funktion smallerArg auf. Diese liefert Variationsmöglichkeiten für den Parameter zurück. Um alle Variationen abzudecken wird am Ende das Kartesische Produkt über die Ergebnisse von smallerArg gebildet. Anschließend werden entsprechend viele Duplikate des Ausgangstyps erzeugt und mit den Ergebnisvektoren aus dem Kartesischen Produkt versehen.

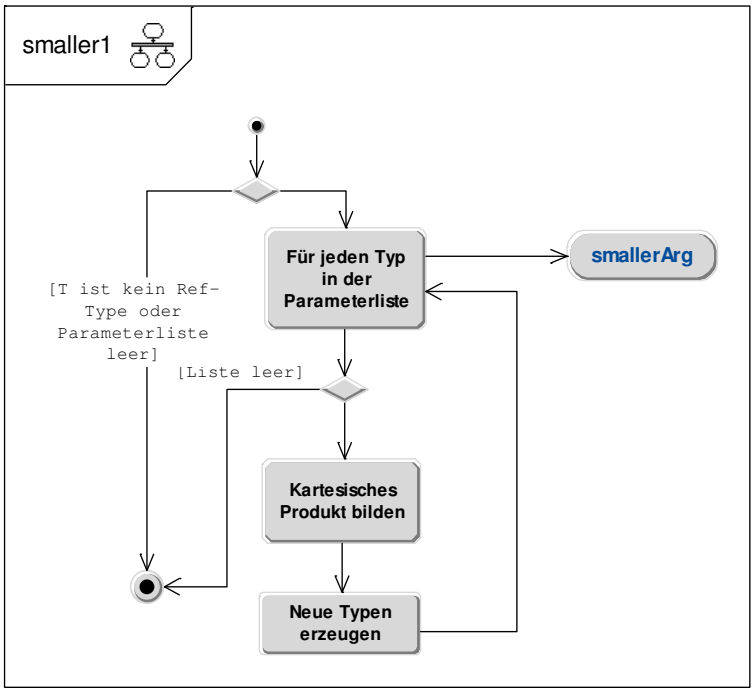


Abbildung 9 – Diagramm smaller1

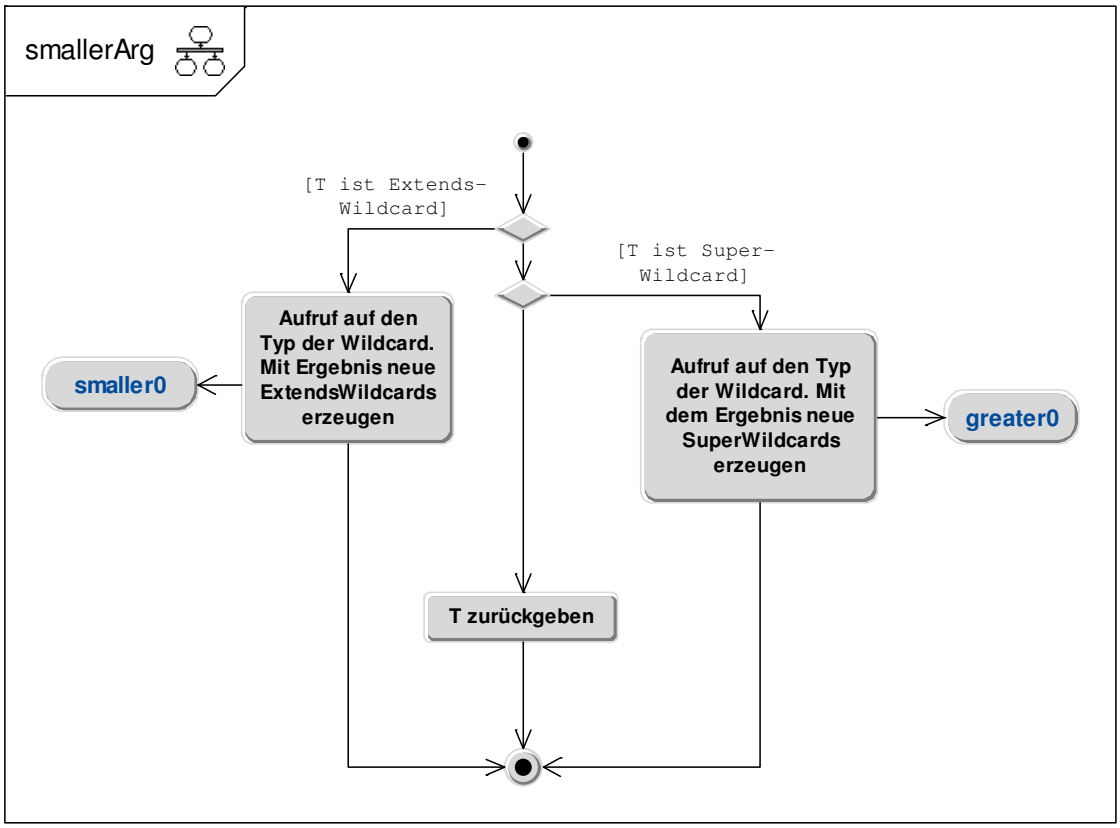
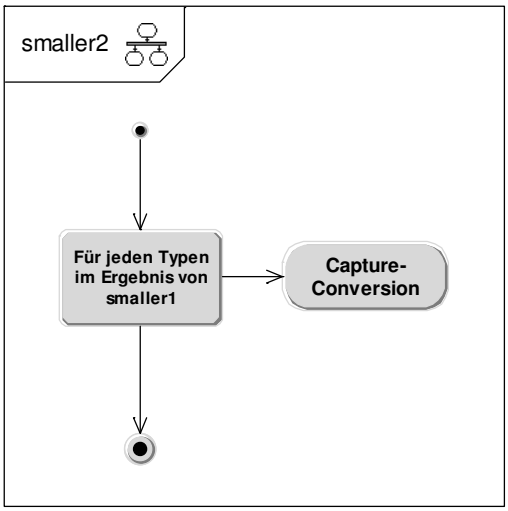


Abbildung 10 – Diagramm smallerArg

smallerArg:

smallerArg ruft je nachdem, ob es sich bei dem übergebenen Typ um eine ExtendsWildcard oder eine SuperWildcard handelt rekursiv die Funktionen smaller0 oder greater0 auf. Im Gegensatz zu greaterArg welches smaller0 und greater0 aufruft, wenn es weder eine Extends- noch eine SuperWildcard ist, macht smallerArg dies nicht. Das rekursive Aufrufen von smaller0 auf beliebige Typen würde zu der in 3.1 beschriebenen unendlich großen Ergebnismenge führen.



smaller2:

smaller2 wendet die CaptureConversion auf alle Elemente aus smaller1 an.

Abbildung 11 – Diagramm smaller2

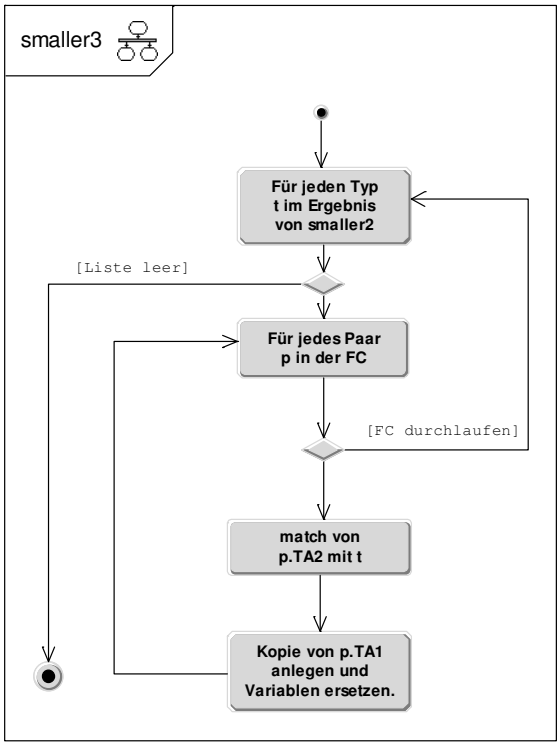


Abbildung 12 – Diagramm smaller3

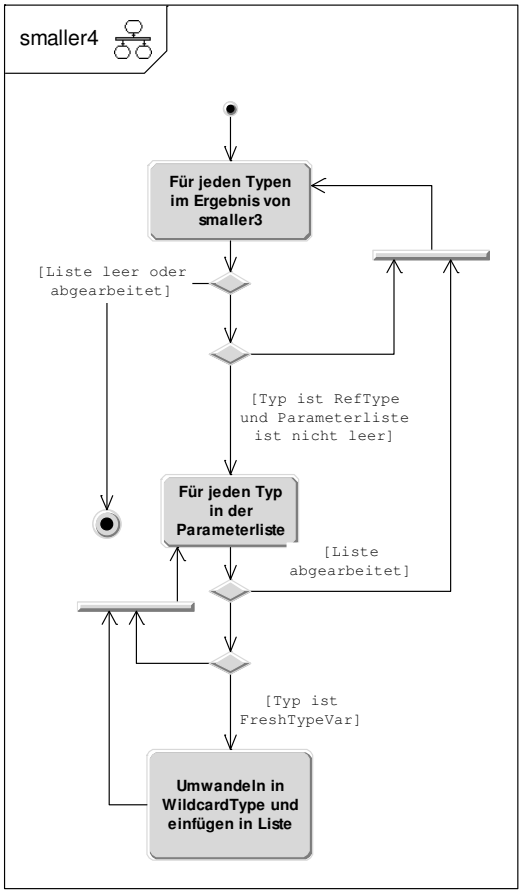


Abbildung 13 – Diagramm smaller4

smaller3:

smaller3 sucht in der FC nach Typen, die durch die Vererbung kleiner sind. Hierzu wird bei allen Paaren die rechte Seite überprüft, ob diese mit einem Typ aus dem Ergebnis von smaller2 gematcht werden kann. Falls ja, dann ist die linke Seite ebenfalls kleiner.

smaller4:

smaller4 wandelt Typen welche über die CaptureConversion in smaller2 erzeugt wurden, und dann in die FC Typen in smaller3 eingesetzt wurden wieder zurück. Beispiel: Teilergebnis von smaller1 sei: {Vector<? extends Integer>}. Durch smaller2 entsteht {Vector<X|^{Integer}>}. smaller3 findet in der FC ein Paar {Stack<A> < Vector<A>}, da Stack von Vector erbt. Smaller3 erzeugt nun { Stack<X|^{Integer}>}. Über smaller4 entsteht wieder { Stack<? extends Integer>}

3.3. Implementierung von cartProduct

cartProduct ist eine Funktion, welche in smaller1 und greater1 verwendet wird. Grund ist folgender. Wenn beispielsweise greater1(Hashtable<String, Integer>) gesucht wird, dann wird zunächst greaterArg(String) = {String, ? extends Object} und greaterArg(Integer) = {Integer, ? extends Number, ? extends Object} ermittelt. In smaller müssen dann alle Kombinationen dieser Ergebnisse ermittelt werden. Gültige Lösungen für greater1(Hashtable<String,Integer>) wären z.B.:

{Hashtable<String, Integer>, Hashtable<? extends Object, Integer>,
Hashtable<String, ? extends Number>, Hashtable<? extends Object, ? extends Number>,
Hashtable<String, ? extends Object>, Hashtable<? extends Object, ? extends Object>}

Da cartProduct für beliebig viele Parameter arbeiten muss, wurde eine rekursive Lösung gewählt. Die Erzeugung des kartesischen Produktes wird durch die Rekursion immer auf eine Produktbildung von zwei Mengen reduziert. Soll nun das Produkt von drei Mengen (durch drei Parameter) gebildet werden, wird zunächst das Produkt der Mengen 2 und 3, und dann in einem weiteren Schritt das Produkt von Menge 1 mit dem Ergebnis aus der vorhergehenden Rekursion gebildet.

3.4. Änderungen in Match

Match ist eine Funktion, welche für die Adapt Regeln, sowie für smaller und greater benötigt wird. Durch Match ist es möglich, Typen die aus der FC entnommen werden die konkreten Variablen zuzuweisen. Beispiel: Die Klasse E<a,b> erbt von D<a,b>. Nun ist folgende Instanz von D vorhanden: D<Integer,Boolean>. Match ermittelt: {a -> Integer, b -> Boolean}. Nun kann das in E eingesetzt werden: E<Integer,Boolean>.

Match wandert rekursiv durch beide Typen durch (den original, und den Typ aus der FC) und vergleicht, wo Einsetzungen vorgenommen wurden. Die Änderung bestand darin, dass Match bisher nur für RefTypes funktioniert hat. Es musste aber auf für WildcardTypes sowie FreshWildcardTypes funktionieren. Deshalb wurde das Interface IMatchable erstellt, welches von diesen Typen implementiert wird.

3.5. Erstellung der neuen Klassen

Im Typsystem des bestehenden Compilers existierte noch keine Basis für die Wildcard Typen. Daher wurden zuerst drei neue Klassen angelegt, welche die drei Wildcard Typen darstellen. Die Klasse *WildcardType* steht für die allgemeine Wildcard und erbt von der Basisklasse *Type*. Von dieser Klasse wiederum erben die Klassen *ExtendsWildcardType* und *SuperWildcardType*. In den Klassen sind jeweils die Eigenschaften welche notwendig sind um die Typen darzustellen. Zusätzlich gibt es noch drei weitere Klassen. Diese *FreshWildcardType* Klassen können vom Benutzer nicht deklariert werden. Im Rahmen der Unifikation werden Wildcard Types in FreshWildcard Types umgewandelt und wieder zurück. Dadurch wird verhindert, dass die Regeln an den falschen Stellen aufgerufen werden. Diese Klassen werden in den Funktionen *smaller* und *greater* erzeugt, werden aber aus der Ergebnismenge gelöscht, bevor diese zurückgegeben wird.

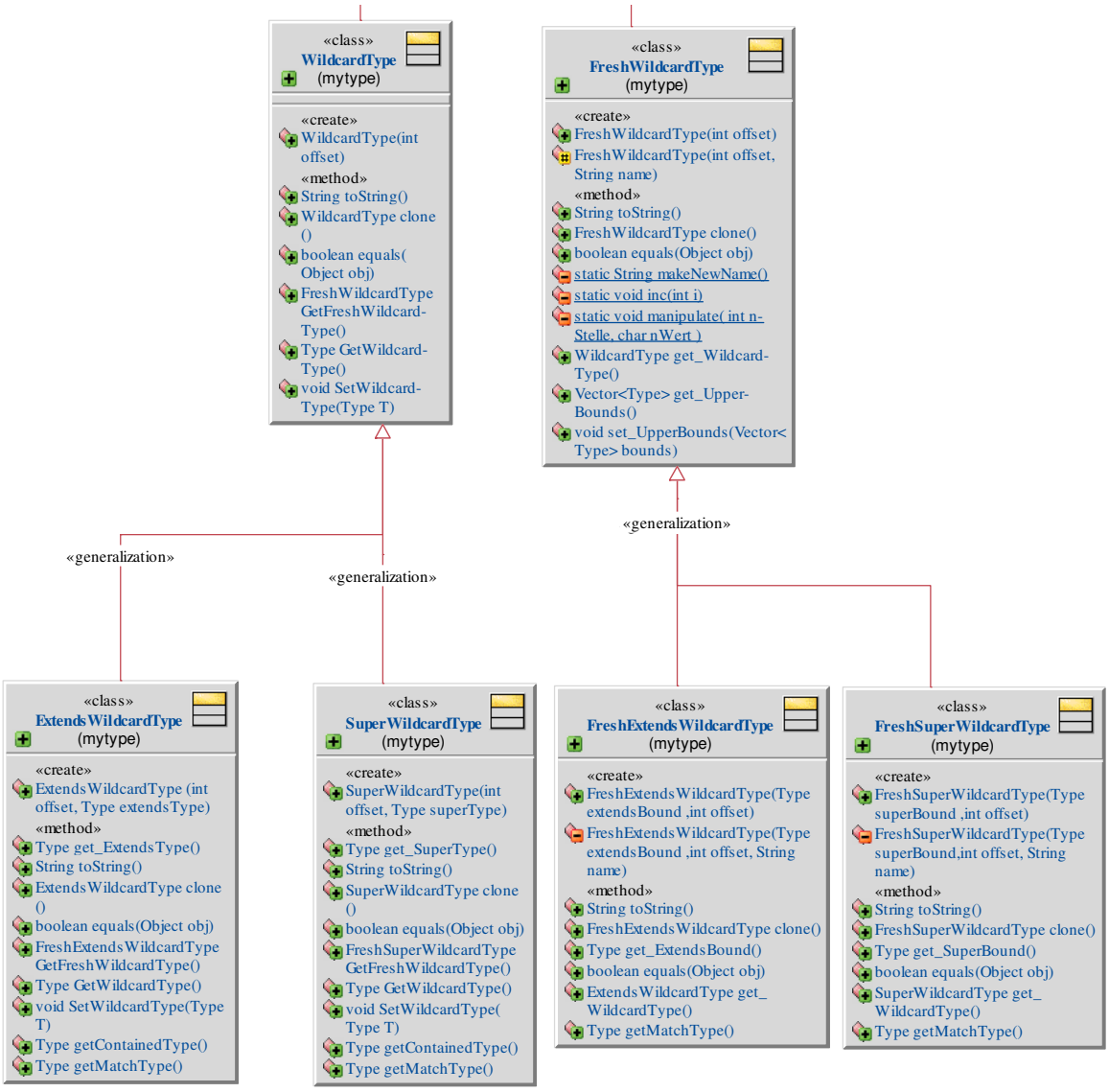


Abbildung 14 – Klassendiagramm der neuen Klassen

3.6. Erweiterung des Parsers

Damit der Parser bereits deklarierte Wildcards erkennt, mussten ein paar zusätzliche Regeln eingebaut, werden. Zuerst musste die Regel für die Parameterliste von Typen so erweitert werden, dass Wildcards als Parameter zugelassen sind. Hierzu wird auf wildcardparameter verwiesen. Wie genau ein wildcardparameter geparst werden muss, wurde in einer 2. Regel festgelegt. Die **fett** geschriebenen Regeln sind neu hinzugekommen.

```
paralist    : IDENTIFIER
{ ... }
            | IDENTIFIER '<' paralist '>'
{ ... }
            | wildcardparameter
{
  ParaList pl = new ParaList();
  pl.getParalist().addElement($1);
  $$ = pl;
}
            | paralist ',' IDENTIFIER
{ ... }
            | paralist ',' IDENTIFIER '<' paralist '>'
{ ... }
            | paralist ',' wildcardparameter
{
  $1.getParalist().addElement($3);
  $$=$1;
}
```

Diese Regel ist neu, und beschreibt, wie ein Wildcardparameter aussieht.

```
wildcardparameter    : '?'
{
  //Luar 29.11.06 Offset auf -1, da keine Angabe vorhanden
  WildcardType wc = new WildcardType(-1);
  $$ = wc;
}
| '?' EXTENDS referencetype
{
  ExtendsWildcardType ewc = new ExtendsWildcardType($2.getOffset(),$3);
  $$ = ewc;
}
| '?' SUPER referencetype
{
  SuperWildcardType swc = new SuperWildcardType($2.getOffset(),$3);
  $$ = swc;
}
```

4. Test der Funktion.

4.1. Modifikationen für Test

Um verschiedene Tests auf einfache weise durchzuführen, wurde von Martin Plümicke vorgeschlagen, den Parser so zu modifizieren, dass dieser direkt zwei Typen einließt. Dadurch ist es möglich zwei Typen direkt zu unifizieren, ohne diese im Programmcode auszuprogrammieren. Die Modifikation war eine neue Regel im Parser. Diese Regel parst die beiden Typen, und speichert diese in einem Pair, welches wiederum in einem Vector<Pair> in der Parser Instanz gespeichert wird. Von dort kann das Pair dann ausgelesen werden. Durch die Rekursion der Regel können beliebig viele Typen geparst werden.

```
compilationunit   : typedeclarations
{ ... }
                  | importdeclarations typedeclarations
{ ... }
                  | packagedeclaration importdeclarations typedeclarations
{ ... }
                  | packagedeclaration typedeclarations
{ ... }
                  | type type compilationunit
{
  this.testPair.add(new Pair($1,$2));
  $$=$3;
}
```

4.2. Test der Funktionen

Für die verschiedenen Tests wurden die Beispiele aus [Plu07] verwendet. Der Grund hierfür war, dass die Beispiele bereits „im Kopf“ berechnet waren, und dadurch die Ergebnisse für einen Vergleich vorhanden sind. Außerdem konnte so überprüft werden, ob die Beispiele in dem Dokument korrekt sind. Das Hauptproblem beim Test ist, dass die Inferierung durch die Wildcards ein hohes Maß an Komplexität gewonnen hat. Deshalb ist es nicht mehr so einfach, zu überprüfen ob die Ausgabe korrekt ist.

5. Ergebnis und Ausblick

Das Ergebnis dieser Studienarbeit ist eine funktionierende Version der Unifikation mit Wildcards. Die Funktionen smaller und greater arbeiteten bei allen getesteten Usecases korrekt.

Ein Problem ist, dass der Bytecode noch keine Wildcards unterstützt. Das muss noch angepasst werden. In diesem Kontext sollte der Bytecode einmal komplett überarbeitet werden, da er diverse Fehler enthält.

VII. Abkürzungsverzeichnis

FC	**Finite Closure**. Menge, welche die Vererbungshierarchie abbildet
TRA	**Typenrekonstruktionsalgorithmus**. Algorithmus, welcher die Typen im abstrakten Syntaxbaum ermittelt.
CC	**Capture Conversion**. Wandelt WildcardTypes in FreshWilcardTypes um.
CVS	**Concurrent Version System**. Ein Versionierungssystem zum verwalten von Softwareständen.

VIII. Quellenangaben

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

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

[Loc06] Andreas Lochbihler. Spracherweiterungen in Java 1.5. Universität Passau, 2006.

[Plu07] Martin Plümicke. Type Inference in Generic Java with Wildcards, not yet published. Berufsakademie Horb, 2007.