STUDIENARBEIT

Berufsakademie Stuttgart – Außenstelle Horb-
Staatliche Studienakademie

Fachrichtung Informationstechnik

Typinferenz in Java

Juni 2004

Eingereicht von:
Thomas Ott
Pferlenstr. 11

78727 Oberndorf-Beffendorf

Firma:
Homag Holzmaschinenbau AG
Homagstrasse 5

72296 Schopfloch

Betreuer:
Prof. Dr. Martin Plümicke

Zusammenfassung

Bisher ist es in der Programmiersprache Java notwendig, alle Typen von Methoden und Attributen explizit anzugeben. Oftmals sind diese Angaben überflüssig und aus dem Kontext zu erschließen.
Diese Studienarbeit implementiert in einen vorhandenen Java-Compiler einen Teil eines Typrekonstruktionsalgorithmuses, beruhend auf der Unifikation von Typen.
Der verwendete Java-Compiler wurde im Rahmen einer Software-Engineering Vorlesung entwickelt und in einer Studienarbeit um generische Typen (Generic-Java, G-JAVA, vgl. [0]) erweitert.

Ehrenwörtliche Erklärung

Ich habe die Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Beffendorf, den 07.06.2004, Thomas Ott

Inhaltsverzeichnis

1. Einleitung

Bisher ist es in der Programmiersprache Java notwendig, alle Typen von Methoden und Attributen explizit anzugeben. Oftmals sind diese Angaben überflüssig und aus dem Kontext zu erschließen.
Diese Studienarbeit implementiert in einen vorhandenen Java-Compiler einen Algorithmus, der solche redundante Typen berechnet.
Derartige Typrekonstruktionsalgorithmen sind bereits aus funktionalen Programmiersprachen bekannt (z.B. SML[1]) und verlangen vom Programmierer nicht, Typen von Bezeichnern explizit zu definieren. Das Verfahren der Typrekonstruktion beruht auf der Unifikation von Typen. Die Implementierung der Unifikation ist der Schwerpunkt dieser Arbeit.

In diesem Zusammenhang spricht man von Typinferenz.
Unter Typinferenz versteht man das Folgern eines Typs für einen Ausdruck, dessen Typ nicht explizit angegeben wurde. Der Typ kann mithilfe des zu implementierenden Algorithmuses berechnet werden.
Der Algorithmus stammt von Martin Plümicke und ist unter [1] spezifiziert.
Typinferenz wird auch in funktionalen Programmiersprachen, wie z.B. SML, eingesetzt. Für den Programmierer bedeutet das eine Erleichterung.

Ein wesentlicher Bestandteil der Typinferenz ist die Unifikation von Typen. Der Unifikations-Algorithmus wurde, mit ein paar Einschränkungen, in dieser Arbeit implementiert.

Die Typinferenz bezieht sich auf Generic Java (G-JAVA, vgl. [1]). Es handelt sich dabei um eine Erweiterung der Sprache Java um parametrisierbare Klassen, teilweise vergleichbar mit C++Templates. Dadurch wird Java lesbarer und zur Laufzeit weniger fehleranfällig durch implizite Typcasts.
Generische Erweiterungen wurden z.B. im „Pizza“ Compiler [2] vorgenommen.
Auch Sun wird *Generics* in der Version 1.5 von Java einbauen[3].
Generics werden hautsächlich für Container Typen verwendet:
Ohne Generics:

```
Liste Temp = new Liste();
Temp.add( new Integer( 4711 ) );
Integer Zahl = (Integer)Temp.elementAt( 0 );
```

Mit Generics:

```
Liste<Integer> Temp = new Liste<Integer>();
Temp.add( new Integer( 4711 ) );
Integer Zahl = Temp.elementAt( 0 );
```

“Integer” wird als *type parameter* oder Typ Parameter bezeichnet.
Die Liste kann im 2. Fall nur noch Objekte vom Typ Integer aufnehmen. Der Compiler kann Verletzungen zur Laufzeit erkennen.

Bei der Implementierung müssen einige Einschränkungen vorgenommen werden. Interfaces werden z.B. nicht berücksichtigt.

[1] SML steht für **S**tandard **M**eta **L**anguage. Es handelt sich um eine funktionale Programmiersprache, die heute v.a. in Forschung und Lehre eingesetzt wird. Vgl. auch [8]

Im Folgenden sind zwei Java-Programme, mit denen beispielhaft gezeigt werden kann, was die Typinferenz berechnen soll, aufgeführt.
Das erste Beispiel ist ein gewöhnliches G-Java Programm mit Typdeklarationen (fett gedruckt), die vom Typinferenz-System berechnet werden sollen. Die Berechnung der Typen macht deren Angabe überflüssig, wie im zweiten Beispiel zu sehen ist.

Beispiel 1:

```
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;
  }
}
```

Beispiel 2:

```
class Matrix extends Vector<Vector<Integer>>
{
  mul(m)
  {
    ret = new Matrix (); i = 0;
    while( i < size() )
    {
      v1 = this.elementAt(i);
      v2 = new Vector<Integer> ();
      int 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; } }
```

2. Analyse des Compilers

Der zu bearbeitende Compiler ist lediglich ein Prototyp, der von vielen Studenten gemeinsam im Rahmen einer Software-Engineering Vorlesung und einer Studienarbeit entwickelt wurde.
Im Rahmen einer Analyse wird festgestellt, was der Compiler kann und was nicht.

- **Allgemeines**
  Die erzeugten CLASS-Dateien sind fehlerhaft und können von einer JVM nicht interpretiert werden. Die Korrektur dieses Fehlers ist Gegenstand einer anderen Studienarbeit [7].

  Ansonsten versteht der Compiler parametrisierte Typen. D.h. Klassen mit Parametertypen können deklariert und instantiiert werden.

  Weiterhin kennt der Compiler keine Packages, Interfaces, Arrays, statische Methoden und for-Schleifen.

- **Rückgabewert bei Funktionsdeklarationen**
  Bei einer Funktionsdeklaration kann als Rückgabewert ein beliebiger Identifier stehen:
```
class Test1
{
  fantasie f1()  {}
}
```

- **Return-Statement**
  Obwohl folgende Funktion ein Return-Statement benötigt, wird fehlerfrei compiliert:
```
class otth3
{
    boolean f1() {}
}
```

- **Mehrfache Attribut- und Methodendeklarationen**
```
class otth1
{
    int a;
    int a; // kein Fehler !!!
    void test() {};
    void test(); // kein Fehler !!!
}
```

- **Mehrfache Variablendeklarationen**
```
class otth1
{
    void test()
    {
       int a;
       int a; // kein Fehler !!!
    }
}
```

- **Vererbung**
```
class otth1 extends fantasie // kein Fehler, obwohl beliebiger
{ … }                        // Identifier
```

- **Konstruktoren**
  Konstruktoren werden zwar vom Parser erkannt, können aber im Semantik-Check nicht weiterverarbeitet werden. Die Folge ist eine NullPointerException:
```
class otth4
{
    otth4() { }
}
```

- **Attributtypen**
  Der Typ eines Attributs (hier: A) kann ein beliebiger Identifier sein. Es muss sich nicht um eine zuvor definierte Klasse handeln:
```
class otth20
{
    A a = null;
}
```

- **Parameter 1**
  Eine parametrierbare Klasse muss eine Instanzvariable des Parameters enthalten:
```
class P<A>
{
  A a; // ohne diese Zeile → Feler!
}
```

- **Parameter 2**
  Eine Parameterklasse muss beim Instanziieren auch initialisiert werden:
```
class P<A>
{
  A a;
}
class otth1
{
  P<otth1> b = null; // ohne  = null → Fehler!
}
```

- **Parameter 3**
  Eine Superklasse kann bei einer Klassendefinition als Parameter nicht nochmals eine Klasse mit Parametern enthalten:
```
class XX<A>
{ A a; }
class YY<B>
{ B b; }
class otth1<B> extends XX<YY<B>> // Fehler
{ }
```

- **Parameter 4**
  Wenn Parameter im Spiel sind, kann eine Basisklasse höchstens eine Superklasse haben. Transitivität (s. Beispiel) ist nicht möglich:

```
class AbstractList<A>
{ … }

class Vektor<A> extends AbstractList<A>
{ … }

class Stapel<A> extends Vektor<A>
{ … }
```

3. Anpassungen des Compilers

3.1. Fehlerbehebung

Einige der in Kapitel 2 beschriebenen Fehler wurden im Rahmen dieser Studienarbeit korrigiert.

Fehler: Mehrfache Attribut- und Methodendeklarationen

Mehrfache Attribut- und Methodendeklarationen sind nicht mehr möglich. Im Compiler werden die Identifier der Klassenmember (Attribute und Methoden) mit zugehörigem Typ in einer Hash-Tabelle gespeichert. In der Funktion `ClassBody.sc_init_hashtable` wird diese Hash-Tabelle aufgebaut. Bevor ein Paar eingefügt wird, wird geprüft, ob ein Identifier bereits in der Hash-Tabelle liegt. Ist dies der Fall, meldet der Compiler einen Fehler.

Fehler: Vererbung

Dieser Fehler wurde behoben. Die Superklasse muss nun eine bereits definierte Klasse sein, d.h. sie muss sich im Klassenvektor (`Sourcefile.KlassenVektor`) befinden. Die Prüfung wird in `ClassBody.sc_init_hashtable` vorgenommen.

Fehler: Parameter 4

Definitionen von Basis- / Superklassen, die Typkonstruktoren enthalten (z.B.: `class otth1<B> extends XX<`**`YY<B>`**`>`) sind nun auch möglich. Im übergebenen Compiler wird fälschlicherweise für jeden Parameter der obersten Ebene überprüft, ob es eine gleichnamige Klasse gibt. Diese Prüfung (`ClassBody.complete_parahashtable`) wurde ausgeschalten und durch die Funktion `ClassBody.istParameterOK` ersetzt. Die neu implementierte Funktion `istParameterOK` überprüft im Rahmen des Semantik-Checks rekursiv den Parameterbaum einer Superklasse. Für `RefType`-Parameter muss eine gleichnamige Klasse existieren. Hat ein RefType-Parameter weitere Parameter (=Typkonstruktor) muss deren Anzahl mit der Anzahl der Parameter in der zugehörigen Klassendefinition übereinstimmen. Handelt es sich um `TyploseVariable`-Parameter, so darf keine Klasse mit diesem Namen existieren (Theoretisch ist dies auf Grund der Funktion `wandleTV2RefType`, s. Kapitel 3.3, auch nicht möglich).
Folgendes Beispiel wird nun akzeptiert:

```
class BB<A>
{ … }
class CC<C, D>
{ … }
class XX<A, X> extends CC<CC<A, B, CC<BB<A>, M>>, BB<BB<BB<A>>>>
{…}
```

Folgendes Beispiel wird als fehlerhaft zurückgewiesen (Fehlerstellen sind **fett** gedruckt):

```
class BB<A>
{ … }
class CC<C, D>
{ … }
class XX<A, X> extends CC<CC<A, CC<A, BB<A>, M>>, BB<Bla<A>>>
{ … }
```

Fehler: Parameter 5
Dieser Fehler geht auf eine fehlerhafte Längenüberprüfung der Parameter zurück und wurde in `Class.para_check` behoben.

3.2. Anpassung des Parsers

Die Java-Spezifikation und somit der Java-Compiler erlauben es nicht, Methoden ohne Rückgabewerte oder Methodenparameter ohne Typen zu definieren.
Folgendes Beispiel soll vom Parser akzeptiert werden. Später werden die Typen davon berechnet.

```
class otth4
{
    …
    f1( a, b )
    {
      …
    }
}
```

Zur Grammatik, d.h. zum JAY-File, wurden folgende Regeln hinzugefügt:

Um Methodenargumente ohne Typ angeben zu können, wurde folgende Rgel hinzugefügt:

```
formalparameter     : variabledeclaratorid
{
  FormalParameter FP = new FormalParameter();
  Type T = new TyploseVariable(“”);
  FP.set_Type( T );
  FP.set_DeclId($1);
  $$=FP;
}
```

Um Methodenargumente auch polymorph zu machen, wurde folgende Regel hinzugefügt:

```
formalparameter     : type variabledeclaratorid
{
  …
}

/* otth: hinzugefuegt */
                    | type ‘<’paralist’>’ variabledeclaratorid
{
  /* Parameterliste setzen */
  $5.set_Paratyp($3.get_ParaList());
  FormalParameter FP = new FormalParameter();
  FP.set_Type($1);
  FP.set_DeclId($5);
  $$=FP;
}
```

Methoden ohne Rückgabewerte werden zunächst immer als Konstruktoren (= Objekte vom Typ `Constructor`) angelegt. Nach dem Parsen und vor dem Semantikcheck (in der Klasse `MyCompiler`) wird über alle Konstruktoren iteriert. Diejenigen Konstruktoren, deren Namen sich von dem Klassennamen unterscheiden, werden zu gewöhnlichen Methoden (= Objekte vom Typ `Method`) „umgewandelt“.

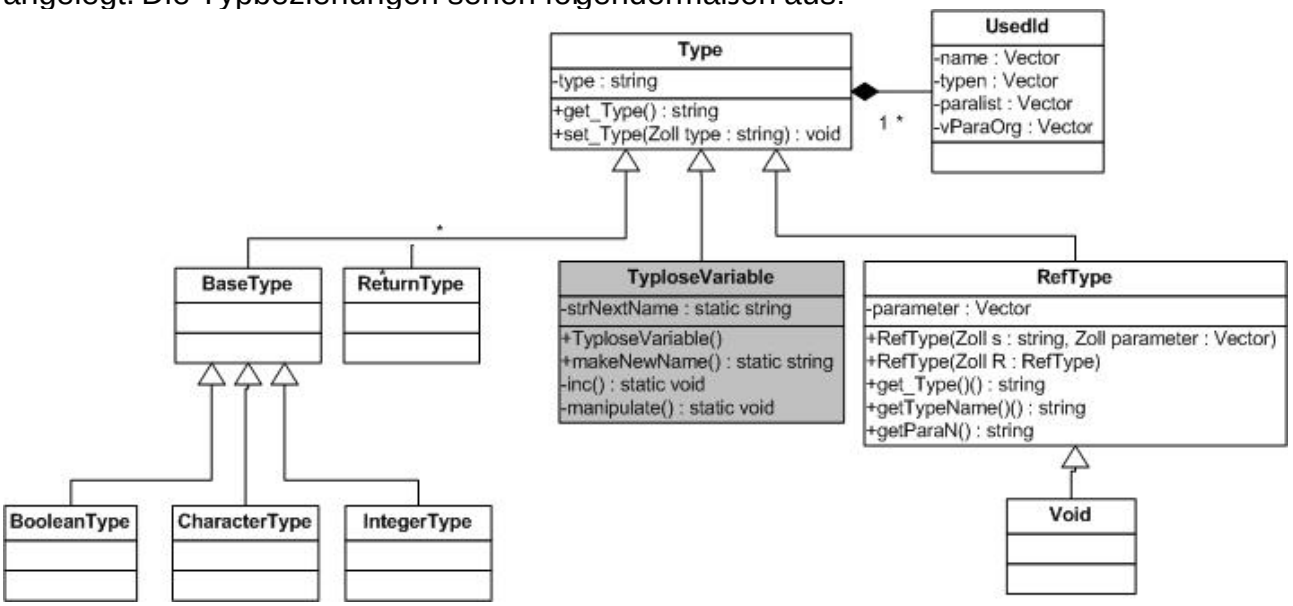
Damit lässt sich folgendes Beispiel realisieren:

```
…
class otth4
{
    …
    f1( Typ1<P1, P2> a, … )
    { … }
}
```

3.3. Anpassung des Designs

Im übergebenen Java-Compiler konnte jeder Typ Parameter enthalten. Das ist ein Designfehler, weil nur Referenztypen, also neu definierte Klassen Parameter enthalten können und nicht etwa primitive Typen.
Um den Fehler zu beheben wurde das Attribut `Vector parameter` von der Klasse `Type` in die Klasse `RefType`, die die Referenztypen repräsentiert, mit allen Konsequenzen, verschoben.

Um Typvariablen speichern zu können, wurde eine neue Klasse `TyploseVariable` angelegt. Die Typbeziehungen sehen folgendermaßen aus:



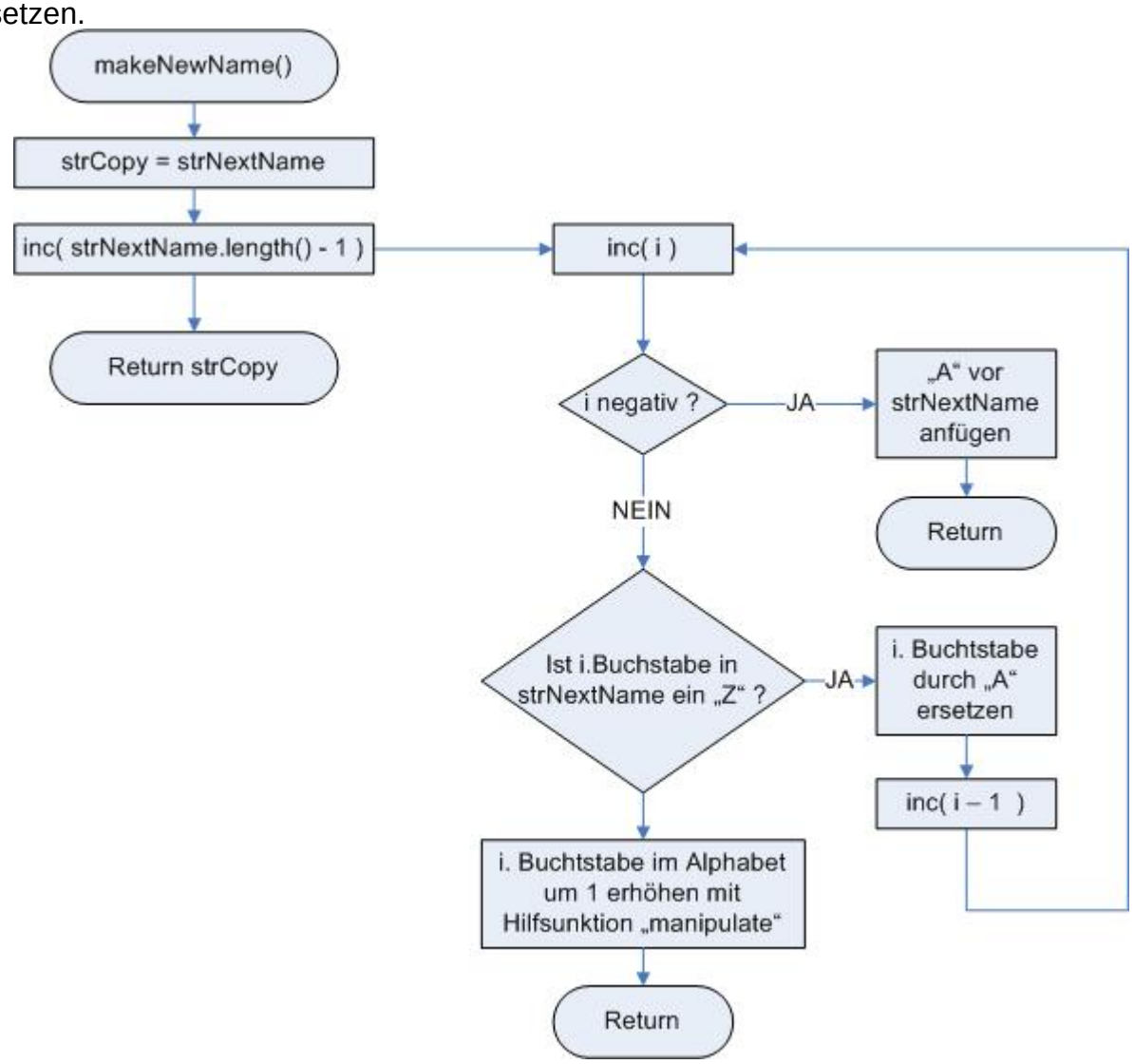
Durchnummerierung

Wenn Funktionsparameter ohne Typen angelegt werden (vgl. Beispiel von oben), dann werden dafür implizit Objekte der Klasse `TyploseVariable` angelegt und „durchnummeriert“.
Die Benennung der Typen funktioniert folgendermaßen: A, B, C, …, Z, AA, AB, AC, …, AZ, BA, BB, …, BZ, CA, …, ZZ, AAA, …
Den Namen, den die nächste Typvariable erhält, wird in der globalen / statischen Variable `TyploseVariable.strNextName` gespeichert.
Wenn der Konstruktor `TyploseVariable` mit einem Leerstring aufgerufen wird, wird automatisch ein neuer Name für das neue Objekt erzeugt, indem im Konstruktor die Funktion `makeNewName` aufgerufen wird. Ansonsten bekommt das Objekt den im Parameter übergebenen Namen.

Funktion: makeNewName
Die Funktion `makeNewName` gibt den in `strNextName` gespeicherten Namen zurück, berechnet den Folgenamen und speichert diesen dann wieder in `strNextName`.
Für das Berechnen werden die beiden Hilfsfunktionen `inc` und `manipulate` benötigt.

Funktion: inc
`inc( i )` erhöht ersetzt den i. Buchstaben des Strings `strNextName` mit dem alphabetischen Nachfolger des i. Buchstabens. Handelt es sich bei dem Buchstaben um ein „Z“, so wird die Stelle i durch ein „A“ ersetzt und der Buchstabe (i-1) um einen Buchstaben „erhöht“, indem rekursiv die Funktion `inc( i – 1 )` aufgerufen wird.
Wird i negativ, so wird vor `strNextName` ein „A“ eingefügt.

Funktion: manipulate
Diese Funktion wird von inc verwendet, um den Buchstaben an der Stelle x im String `strNextName` durch einen Buchstaben, der zum übergebenen ASCII-Wert gehört, zu ersetzen.



Anpassung der Parameterliste

Wie in [4] beschrieben, werden die Parameter eines Typs / einer Klasse in einem Vektor des zugehörigen Objekts gespeichert. In dem Vektor werden die eigentlichen Parameter gespeichert. Im übergebenen Compiler wurden darin stets Objekte vom Typ `Type`

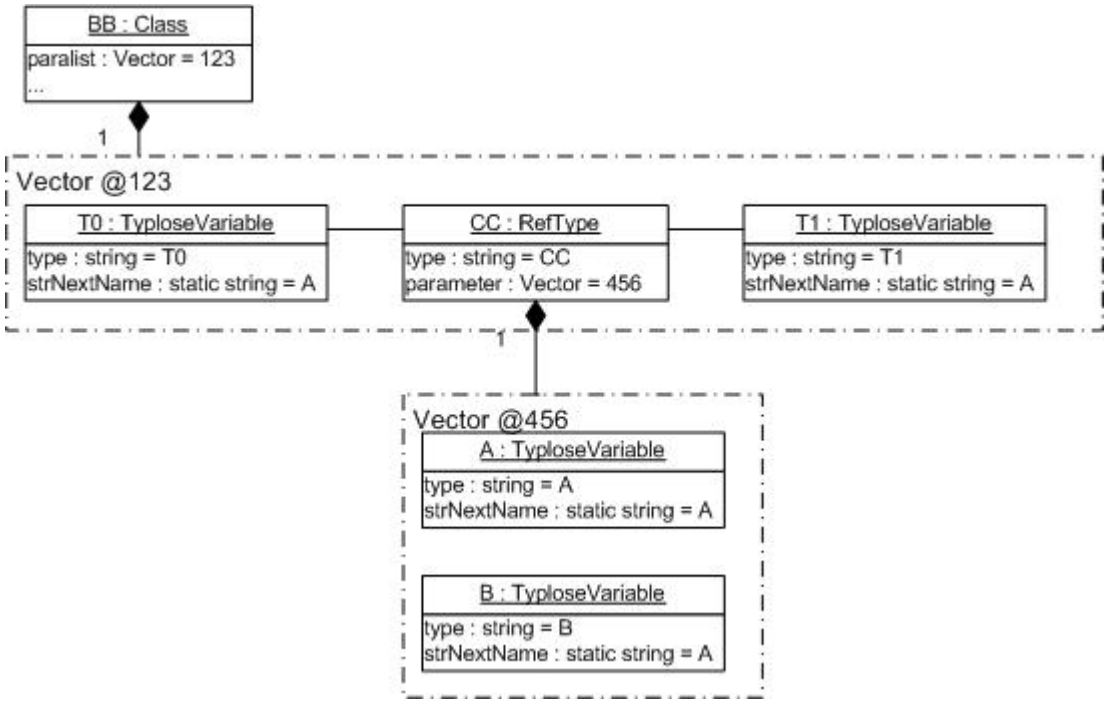
gespeichert. Nach der Umstrukturierung, d.h. nach der Verschiebung von Vector Parameter in die Klasse `RefType` ist es notwendig, dass im Parametervektor entweder Objekte vom Typ `RefType` oder vom Typ `TyploseVariable` gespeichert werden. Um Typkonstruktoren zu speichern, also Typen mit Parameter, müssen `RefType`'s angelegt werden und um Typvariablen zu speichern müssen `TyploseVariablen` angelegt werden.
Dazu werden die entsprechenden Grammatikregeln in der Datei `JavaParser.jay` wie folgt geändert (vgl. fett gedruckte Passagen):

```
paralist : IDENTIFIER
  {
    ParaList pl = new ParaList();
    pl.paralist.addElement( new TyploseVariable($1.getLexem()));
  }
  | IDENTIFIER '<' paralist '>'
  {
    ParaList pl = new ParaList();
    RefType t = new RefType( $1.getLexem() );
    t.set_ParaList( $3.get_ParaList() );
    pl.paralist.addElement(t);
    $$ = pl ;
  }
  | paralist ',' IDENTIFIER
  {
    $1.paralist.addElement(new TyploseVariable($3.getLexem()));
    $$=$1 ;
  }
  | paralist ',' IDENTIFIER '<' paralist '>'
  {
    RefType t = new RefType( $3.getLexem() );
    t.set_ParaList( $5.get_ParaList() );
    $1.paralist.addElement(t);
    $$=$1;
  }
```

Die Klasse `ParaList` wird temporär benötigt, um den „Parameterbaum" aufzubauen.
Das folgende Beispiel verdeutlicht die Speicherung der Parameter.
Beispiel:

```
class … extends BB< T0, CC<A, B>, T1 >
```

Der Parametervektor / Parameterbaum der Klasse `BB` sieht, als Objektdiagramm dargestellt, folgendermaßen aussehen:



Die Funktion `wandleTV2RefType`

Objekte vom Typ `TyploseVariable` werden also immer dann angelegt, wenn in einer Parameterliste ein Identifier ohne nachfolgendes „<“ auftaucht. Für gewöhnliche Objekte als Parameter werden somit auch `TyploseVariablen` angelegt, obwohl solche Objekte durch `RefType`'s mit `ParaList = null` repräsentiert werden. Deshalb sucht die Funktion `wandleTV2RefType` in jeder Superklassen-Parameterliste des Javaprogramms rekursiv nach Parametern, die keine weiteren Parametern in spitzen Klammern enthalten. Das Ergebnis ist eine Submenge der Objekte vom Typ `TyploseVariable`. Existiert für eine `TyploseVariable` eine gleichnamige Klasse im Javaprogramm, so wird dieses Objekt durch ein neu instantiiertes `RefType`-Objekt mit gleichem Namen und `ParaList = null` ersetzt.

Beispiel:

```
class M { … }
class XX<A, X> extends CC<CC<A, CC<A, BB<A>>>, BB<BB<M>>> {…}
```

Für die Identifier „A“ werden `TyploseVariablen` angelegt, für den Identifier „M“ wird zunächst auch eine `TyploseVariable` angelegt, die aber, da eine Klasse „M“ exisitiert, in einen `RefType` mit `ParaList = null` umgewandelt wird.

Die Funktion `wandleTV2RefType` wird direkt nach dem Parsen und vor dem Semantikcheck aufgerufen.

3.4. Kommandozeilenparameter

Ein Compiler ist ein sehr komplexes Programm, das aus verschiedenen Phasen besteht (z.B. Scannen, Parsen, Semantik „Checken“, …). Um die Übersicht nicht zu verlieren, wurde eine „Debug-Strategie“ eingeführt. Über eine Zahl (= Debug-Level), die als Kommandozeilenparameter übergeben wird, kann der Anwender festlegen, aus welchem Bereich Debug-Infos angezeigt werden sollen.

Die Syntax für den Compiler-Aufruf sieht folgendermaßen aus:

```
Useage: java MyCompiler <File> [Debug-Level]
```

```
Debug-Level (optional):
- []   --> keine Debug-Infos
- 0    --> alle vorhandenen Debug-Infos
- 1    --> allgemeine Debug-Infos
- 2    --> Scanner-Debug-Infos
- 3    --> Parser-Debug-Infos
- 4    --> Semantik-Check-Debug-Infos
- 5    --> Typenlose Methoden
- 6    --> Unifikation1
```

Das Debug-Level wird in einer statischen Variablen festgehalten: `MyCompiler.nDebug`. Mit der Funktion `MyCompiler.Debug` kann eine Debug-Information ausgegeben werden. Dazu muss der Funktion der Ausgabetext und das Debug-Level übergeben werden.

Vorerst kann der Unifikationsalgorithmus über das Debug-Level „6“ getestet werden. Im Rahmen der Typinferenz wird der Algorithmus dann implizit aufgerufen.

4. Der Unifikationsalgorithmus

4.1. Grundbegriffe

In diesem Abschnitt werden einige Begriffe erklärt, die für das Verständnis der Unifikation und der zugehörigen Literatur wichtig sind. Vgl. auch [5].

Polymorphie

Die Polymorphie lässt sich anhand eines Beispiels erklären.

```
Class Liste<A>
{
  A Element;
  void add( A a );
}
…
Liste<Integer> Temp = new Liste<Integer>();
Temp.add( new Integer( 4711 ) );

Liste<Float> Temp = new Liste<Float>();
Temp.add( new Float( 3.14 ) );
```

Dieselbe Funktion `add` kann auf unterschiedliche Typen angewandt werden. Diese Funktion nennt man polymorph. Dieser Begriff bezieht sich auf die Parameter, die unterschiedlich (gr. „poly“) gestaltet (gr. „morph“) sein können.
`Liste<A>` nennt man einen polymorphen Typ.

Typvariable

In [6] werden Typvariablen wie folgt definiert:
„Eine Typvariable ist eine Variable für den Typ einer im Programm vorkommenden Variablen bzw. eines dort vorkommenden Ausdrucks. Die Werte der Typvariablen sind Teilmengen von AT.“
AT bezeichnet die Menge alle im Programm vorkommenden Typen.
z.B.: AT = { int, bool, …, Klasse1, Klasse2, … }
Im oberen Beispiel (`class List<A>, add( A a ),` …) ist „A“ eine Typvariable.

Typinferenz

„Die Typinferenz ist die Schlussfolgerung, durch die der Typ eines Ausdrucks ermittelt wird …“ [5].
Beispiel:

```
…
Integer mal( Integer Zahl1, Integer Zahl2 );
…
int Zahl3 = mal( 2, x );
…
```

Anhand der Funktionssignatur kann der Typ von x ermittelt werden, falls dieser unbekannt ist: `Integer`.

Typkonstruktoren

In diesem Kontext werden Typen mit Parametern bzw. die Operation zum Erzeugen der Typen als Typkonstruktoren bezeichnet.

Schlussregeln

Zur Beschreibung des Unifikationsalgorithmuses werden Schlussregeln verwendet. Mit einer Schlussregel können anhand von bereits festgestellten Aussagen weitere Aussagen geschlossen werden.
Eine Schlussregel hat folgende Gestalt:

$$\frac{\textbf{Voraussetzung1, Voraussetzung2, ...}}{\textbf{Schluss}}\textbf{(Annahme)}$$

Sie bedeutet, dass unter den Vorraussetzungen und der Annahme der logische Schluss folgt. Äquivalente Darstellung:

$$\textbf{Voraussetzung1} \wedge \textbf{Voraussetzung2} \wedge \textbf{, ...,} \wedge \textbf{Annahme} \rightarrow \textbf{Schluss}$$

Unifikationsalgorithmus

Bei der Unifikation werden zwei Ausdrücke „gleichgemacht“, indem eine Substitution (Ersetzung) auf beide Terme angewendet wird. Diese „gleichmachende“ Substitution wird dann Unifikator genannt. Ziel des Unifikationsalgorithmuses ist es eine solche Substitution zu finden.
Beispiel:

- Klassendefinition: `class Stack<A> extends Vektor<A>`
- Typausdruck1: `Stack<Stack<Interger>>`
- Typausdruck2: `Vektor<Vektor<A>>`

→ `unify( Typausdruck1, Typausdruck2 ) = { A → Integer }`
Das Ergebnis ist der Unifikator $s = \{A \rightarrow Integer\}$. Wendet man den Unifikator / die Substitution auf beide Ausdrücke an, so werden sie, unter Berücksichtigung der Ableitung, gleichgemacht.

4.2. Der Unifikationsalgorithmus für Java

Für die Sprache Java wurde von Martin Plümicke ein Unifikationsalgorithmus entwickelt, der Grundlage des Typinferenzsystems von Java ist. Der Algorithmus basiert auf Arbeiten von Herbrand, Martelli und Montanari (Literaturverweise in [0]).
In [0] ist der Algorithmus beschrieben. In [1] sind Grundlagen und Definitionen, die für den Algorithmus wichtig sind, beschrieben.
In dieser Arbeit wird nur eine vereinfachte Form des Algorithmuses implementiert. Das Ergebnis schränkt sich auf Grund der Vereinfachung auf einen Unifikator ein. Interfaces wurden vollkommen weggelassen.
Im Folgenden wird die Form des Algorithmuses nach [0], die implementiert wurde, beschrieben.
Um den Algorithmus zu verstehen, werden zuerst einige Begriffe definiert oder an einem Beispiel veranschaulicht (nach [1]).

Begriffe / Definitionen / Anmerkungen

Typvariablen:
TV
Beispiel: $TV = \{A, B, C, D, ...\}$

Menge aller Typterme über einem Rangalphabet ?:
$T_\Theta(TV)$
Beispiel: $T_\Theta(TV) = \{Vektor < A >, Vektor < Ineter >, Vektor < Vektor < Integer >>, ...\}$

Rangalphabet:
Θ
Beispiel: $\Theta = \{\{Integer, Object\}_0, \{Vector, Stack\}_1, \{Matrix, Pair\}_2\}$
Bemerkung: Alle Klassen, die dieselbe Anzahl an Parameter haben, werden in einer Menge zusammengefasst. `Matrix<A, B>` und `Pair<A, B>` haben beispielsweise jeweils zwei Parameter.

Typvariablen eines Typs:
$TVar : T_\Theta(TV) \rightarrow r(TV)$
Beispiel: $TVar(Pair < A, B >) = \{A, B\}$

Substitution, Unifikator, „type unifier“:
s

Hierarchie von Java-Klassen:
$(\text{Klasse1} < a_1, a_2, ..., a_n >, \text{Klasse2} < a_1, a_2, ..., a_n >) \in \leq \leftrightarrow \text{Klasse1} < a_1, a_2, ..., a_n > \text{ extends Klasse2} < a_1, a_2, ..., a_n >$

Bemerkung: Die Relation = spiegelt genau die Vererbungshierarchie von Java wieder. Über dieser Menge wird eine Hülle gebildet, um die reflexiven und transitiven Eigenschaften von abgeleiteten Klassen aufzunehmen: =*

Definition: Closure of the type term ordering (nach [1])
Sei $T_\Theta(TV)$ eine Menge an Typtermen und = eine Typtermordnung auf $T_\Theta(TV)$. Die Hülle der Typtermordnung = (bezeichnet als =*) ist gegeben als die kleinste Ordnung mit folgenden Bedingungen:

- Falls $(q, q') \in T_\Theta(TV) \times T_\Theta(TV)$ ein Element der reflexiven und transitiven Hülle von = ist, folgt: $q \leq^* q'$
- Falls $q_1 \leq^* q_2'$ gilt, folgt: $s_1(q_1) \leq s_2(q_2)$ für alle Substitutionen s_1, s_2, die der folgenden Bedingung genügen: $s_1(A) = s_2(A)$ für alle $A \in T_\Theta(TV)$

Bemerkung: Die Menge =* macht die Menge = reflexiv und transitiv. Im ersten Schritt werden die Paare von = übernommen. Im zweiten Schritt werden alle möglichen Substitutionen eines bereits in der Menge =* liegenden Paares zur Menge =* hinzugefügt. Dadurch wird die Menge (meistens) unendlich groß.
Beispiel:
```
class Stack<A> extends Vektor<A>; class List<A>
=  = { (Stack<A>, Vektor<A>) }
=* = { (Stack<A>, Vektor<A>), (Stack<List<A>>, Vektor< List<A>>)
       (Stack<List<List<A>>>, Vektor< List<List<A>>>), … }
```

Definition: Finite Closure von = (nach [1])
Sei = eine Typtermordnung und =* ihre Hülle.
Dann ist die *finite closure* ***FC(=)*** von = folgendermaßen definiert:

$$FC(=) := \{\overline{q \leq^* q' \mid q \leq q'}\} \cup \{q \leq^* q'' \mid q \leq q'\}$$

Bemerkung: Die Menge FC ist ein Teilmenge von = und wird für die Adapt-Regel benötigt. Diese Definition ist, wie im Laufe der Arbeit festgestellt wurde, nicht ganz korrekt und muss noch geändert werden.

Definition: Solved form
Eine Menge von Gleichungen ist in *solved form*, falls

$$E_q = \{ a_1 \dot{=} q_1, \ldots, a_n \dot{=} q_n \}$$

wobei $a_1, \ldots, a_n \in$ TV immer paarweise verschiedene Variablen sind,
$q_1,...,q_n \in T_\Theta(TV)$ und
für alle i, j $\in \{1,...,n\}$ $a_i \notin TVar(q_j)$.

Unifikationsalgorithmus (nach [1], vereinfacht)

Eingabe: $E_q = \{ q_1 <\cdot q_1,...,q_n <\cdot q_n \}$
Als Eingabe dient die Menge Eq. Die Menge Eq ist eine Menge von Paaren, die Gleichungen von Typausdrücken repräsentieren, die unifiziert werden sollen. Sie ist am Anfang mit mindestens einem Paar gefüllt.
Es gibt zwei Arten von Gleichungen:

1. $(q_1 <\cdot q_2)$ bedeutet, dass die beiden Typen unifiziert werden sollen, so dass $s(q_1) <^* s(q_2)$. Am Anfang stehen nur Paare diesen Typs in Eq.
2. $(q_1 \dot{=} q_2)$ bedeutet, dass die beiden Typen unifiziert werden sollen, so dass $s(q_1) = s(q_2)$

Ausgabe: Unifikator s

Berechnung:
Jede der folgenden Regeln wählt willkürlich ein Paar aus der Menge Eq aus und verändert dies. Die Regeln werden sooft angewandt, bis für jedes Paar in Eq keine Regel mehr anwendbar ist.

- **REDUCE1**-Regel
  $$\frac{Eq \cup \{C\langle q_1,...q_n\rangle <\cdot D\langle q'_1,...q'_n\rangle\}}{Eq \cup \{q_{p(1)} \doteq q'_1,...,q_{p(n)} \doteq q'_n\}}$$
  wobei gilt:
  - $C\langle A_1,...A_n\rangle \le D\langle A_{p(1)},...,A_{p(n)}\rangle$
  - $A_1,...A_n \in TV$
  - π ist eine Permutation

- **REDUCE2**-Regel
  $$\frac{Eq \cup \{C\langle q_1,...q_n\rangle \doteq D\langle q'_1,...q'_n\rangle\}}{Eq \cup \{q_{p(1)} \doteq q'_1,...,q_{p(n)} \doteq q'_n\}}$$

- **ERASE**-Regel
  $$\frac{Eq \cup \{q \doteq q'\}}{Eq} q = q'$$

- **SWAP**-Regel
  $$\frac{Eq \cup \{q \doteq A\}}{Eq \cup A \doteq q} q \notin TV, A \in TV$$

- **ADAPT**-Regel
  $$\frac{Eq \cup \{D\langle q_1,...q_n\rangle < \cdot D\langle q'_1,...q'_m\rangle\}}{Eq \cup \{q_1 \doteq l'_{1'},...,q_n \doteq l'_n\}}$$
  Dabei gibt es $\bar{q}'_1,...\bar{q}'_m$ mit
  - $D\langle A_1,...A_n\rangle \leq *D'\langle \bar{q}'_1,...,\bar{q}'_m\rangle \in FC(\leq)$ und
  - $D'\langle q'_1,...,q'_m\rangle = D'\langle \bar{q}'_1,...,\bar{q}'_m\rangle[A_i \mapsto l'_i 1 \leq i \leq n)$

- **SUBST**-Regel
  $$\frac{Eq \cup \{A \doteq q\}}{Eq[A \mapsto q] \cup \{A \doteq q\}} \text{A kommt in Eq vor, aber nicht in } q$$
  Diese Regel muss als letzte angewandt werden.

Nach der Anwendung der Regeln muss noch geprüft werden, ob sich die Menge Eq in „solved form“ befindet.

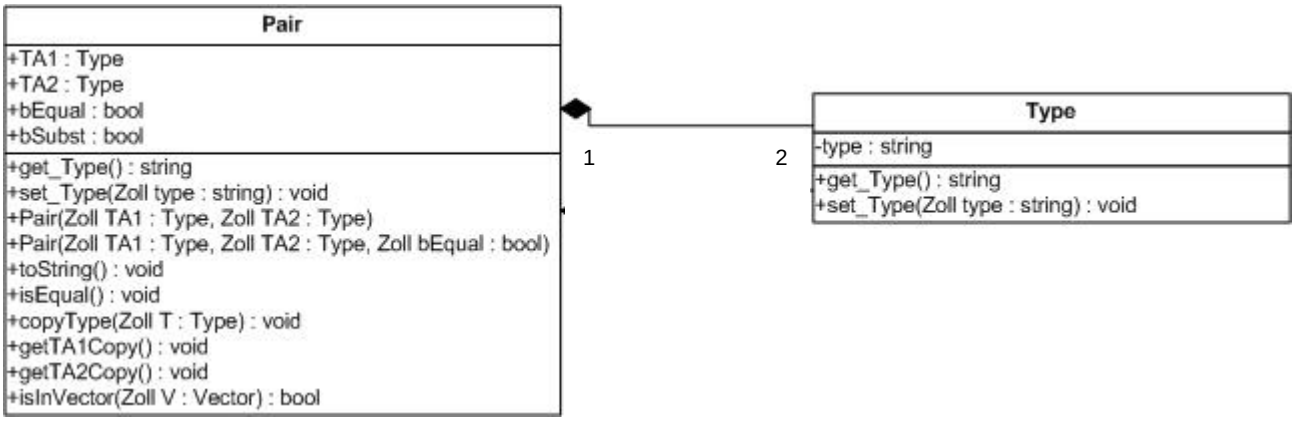
4.3. Implementierung der Unifikation

4.3.1. Hilfsfunktionen

Bei der Unifikation müssen sehr viele Operationen auf Mengen ausgeführt werden. Verschachtelte `for`-Schleifen und Rekursionen sind ein wichtiges Hilfsmittel.

Die Datenstruktur / Klasse `Pair`

Um die Paare in der Menge Eq speichern zu können, wurde eine neue Klasse `Pair` definiert.



Diese Klasse kann für die linke und für die rechte Gleichungsseite jeweils einen Typausdruck aufnehmen. Diese Attribute (`TA1, TA2`) sind vom Typ `Type`.

Funktion: `Pair.toString`

Diese Funktion wandelt ein Paar in einen String um. Sie wird neben Debug-Zwecken verwendet, um die Gleichheit zweier Paare festzustellen. Typen, die eine Parameterliste haben werden über die Funktion `RefType.get_Type()`[4] serialisiert.

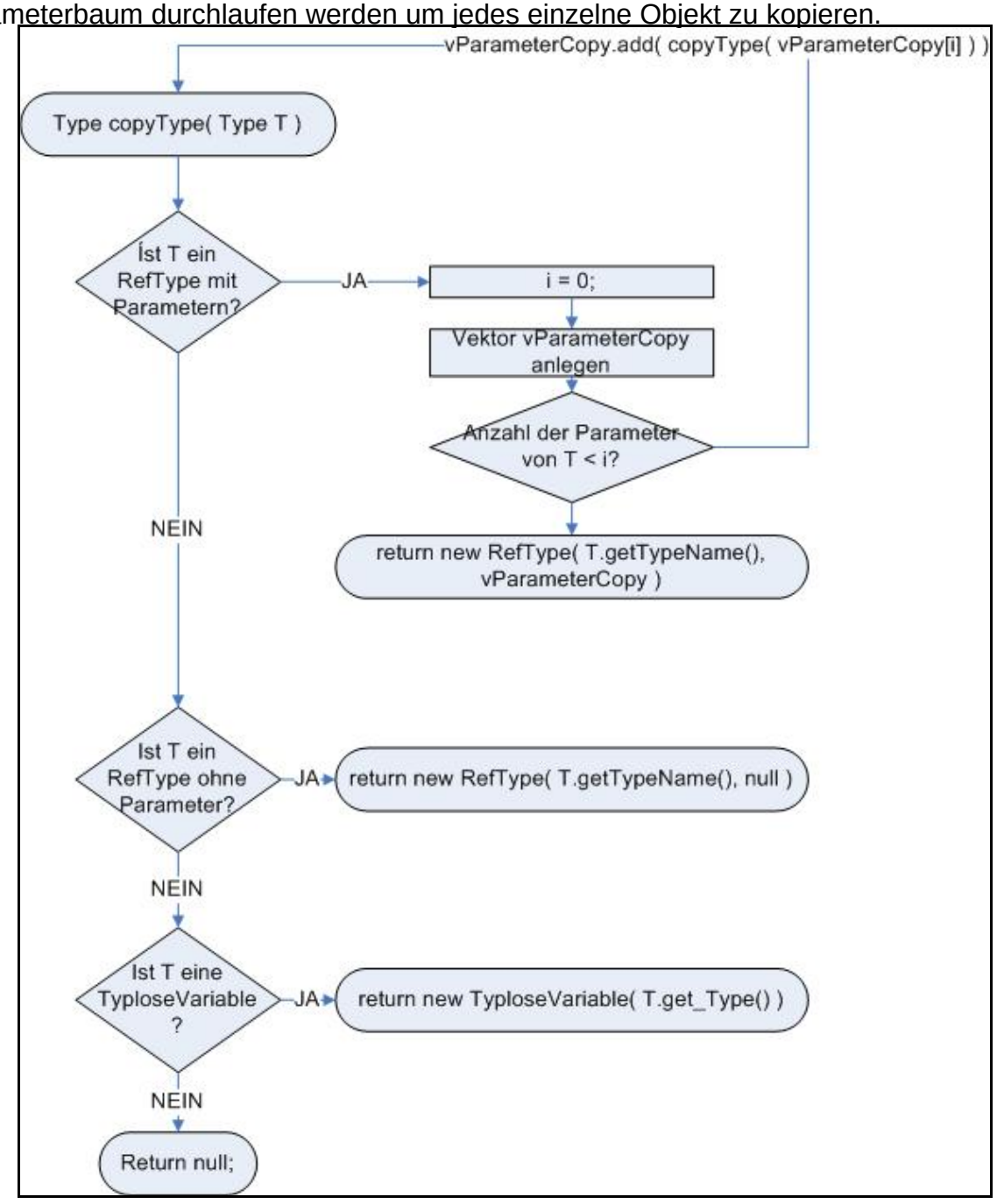
Beispiel: `(Matrix< List< B > >, Vektor< Vektor< List< Objekt > > >)`

Funktion: `Pair.isEqul`

Diese Funktion prüft, ob der linke Typausdruck (TA1) gleich ist wie der rechte Typausdruck (TA2). Die Funktion wandelt den linken Typ und den rechten Typ in einen String um (Funktion: `RefType.get_Type(), Type.get_Type()`) und vergleicht diese.

Funktion: `Pair.copyType`

Während der Unifikation müssen Paare erzeugt und kopiert werden. Es dürfen aber nicht nur die Referenzen kopiert werden. Deshalb muss beim Kopieren rekursiv der Parameterbaum durchlaufen werden um jedes einzelne Objekt zu kopieren.



Funktion: `Pair.getTA1Copy, Pair.getTA2Copy`
Diese Funktionen kopieren nur den Typausdruck1 bzw. Typausdruck2 des Paares, indem die Funktion `copyType` aufgerufen wird.

Funktion: `Pair.isInVector`
Diese Funktion überprüft, ob das aktuelle Paar in einem der Funktion übergebenen Vektor liegt. Die Funktion durchläuft den übergebenen Vektor und prüft für jedes Element, ob es sich um ein `Pair` handelt. Ist dies der Fall, so wird über die Funktion `toString` einen Stringverleich mit dem aktuellen Paar und dem Vektorelement gemacht. Stimmen die Strings überein, wird `true` zurückgegeben.

Funktion: `printMenge`
Die Menge Eq oder die Menge FC werden als Vektoren von Paaren gespeichert. Um diese für Debug-Zwecke auszugeben, dient die Funktion `printMenge`. In einer Schleife wird der Vektor durchlaufen und über die Funktion `Pair.toString` wird jedes Paar-Element an einen `String` angehängt, der am Schluss ausgegeben wird.

4.3.2. Die Funktion `makeFC`

Die Menge FC ist eine Teilmenge von =* und wird für die Adapt-Regel benötigt. Sie wird vor der eigentlichen Unifikation mit der Funktion **`makeFC`** berechnet (s. Kapitel 4.2: Definition von FC). Am Ende des Kapitels wird darauf hingewiesen, dass die Menge FC noch nicht vollständig implementiert wurde, was die Menge der zulässigen Java-Programme einschränkt.
Ein Beispiel soll verdeutlichen, wie die Menge FC aussieht.

In einem Java-Programm seien folgende Klassen definiert:

```
class AbstractList<A> { … }
class Vektor<A> extends AbstractList<A> { … }
class Matrix<A> extends Vektor<Vektor<A>> { … }
class ExMatrix<A> extends Matrix<A> { … }
```

Die zugehörige Menge FC, die von `makeFC` berechnet wird, sieht so aus:

```
FC = { (Vektor< A >, Vektor< A >),
       (Vektor< A >, AbstractList< A >),
       (Matrix< A >, Matrix< A >),
       (Matrix< A >, Vektor< Vektor< A > >),
       (ExMatrix< A >, ExMatrix< A >),
       (ExMatrix< A >, Matrix< A >),
       (Vektor< Vektor< A > >, Vektor< Vektor< A > >),
       (Vektor< Vektor< A > >, AbstractList< Vektor< A > >),
       (Matrix< A >, AbstractList< Vektor< A > >),
       (ExMatrix< A >, Vektor< Vektor< A > >),
       (ExMatrix< A >, AbstractList< Vektor< A > >) }
```

Die Menge FC bzw. die Obermenge =* wurden nicht genau nach der Definition implementiert, da z.B. =* nicht endlich ist und die Definition von FC noch nicht vollständig. Das Grundprinzip ist, aus
`class Matrix<A> extends Vektor<Vektor<A>>` und
`class Vektor<A> extends AbstractList<A>` die Beziehung
`( Matrix<A>, AbstractList< Vektor<A>> )` herzustellen.

Die Funktion `makeFC` besteht aus zwei Teilen.

Teil 1 von `makeFC`:

Im ersten Teil wird die komplette Menge = zur Menge FC hinzukopiert.
Sämtliche, in einem Java Programm definierten Klassen, werden vom Parser in dem Vektor `KlassenVektor` (Attribut der Klasse `SourceFile`) gespeichert.
Dieser Vektor wird durchlaufen.
Für jede Klasse, die eine Superklasse hat, werden zwei neue Paare gebildet.
Eine Klasse hat eine Superklasse, falls das Attribut `Class.superclassid` auf ein Objekt vom Typ `UsedID` verweist. In `UsedID` sind Name und Parameter der Superklasse gespeichert.
Folgende Paare werden gebildet und der Menge FC hinzugefügt:

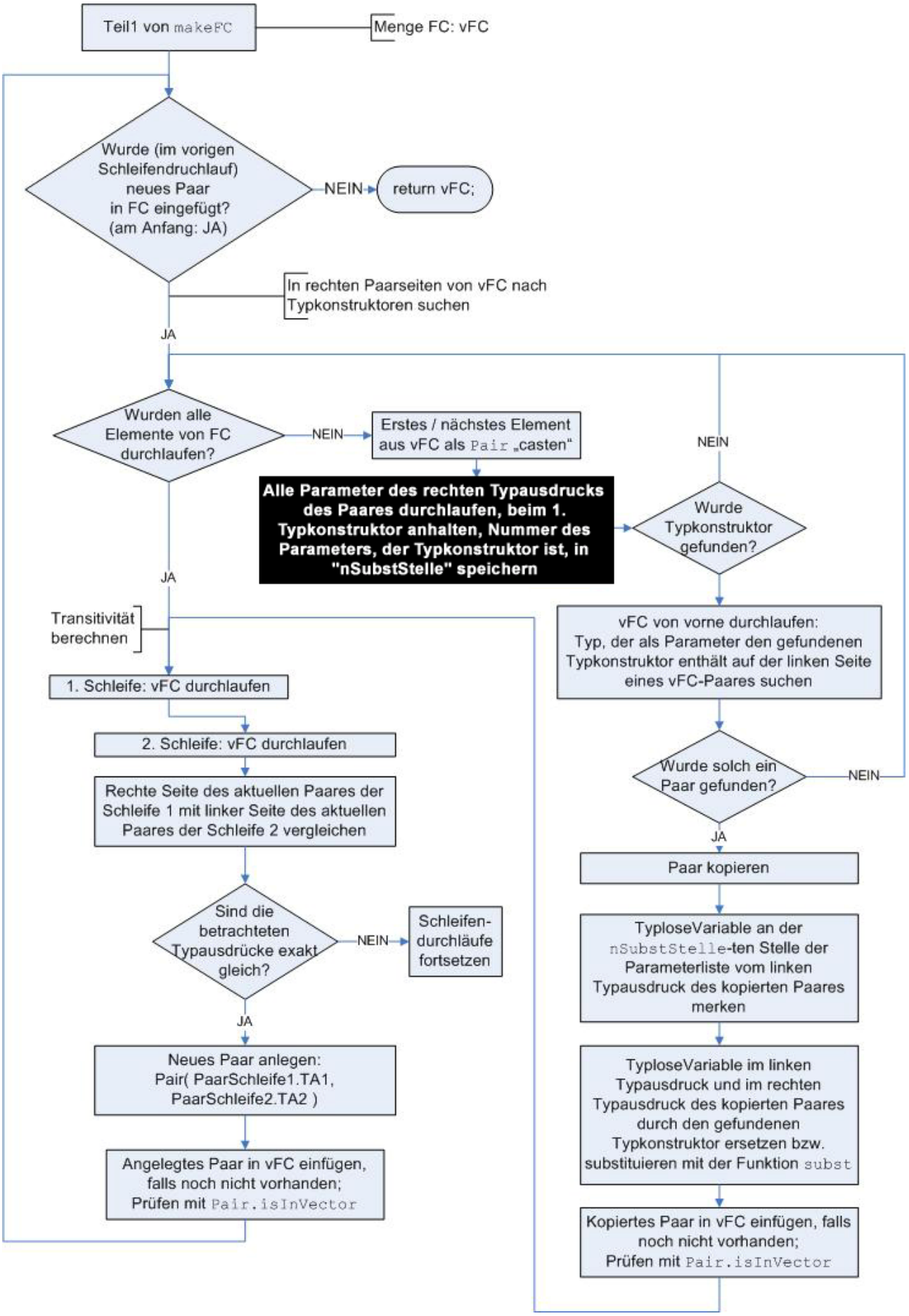
```
Pair P1 = new Pair(
 new RefType( tempKlasse.get_classname(), tempKlasse.vParaOrg ),
 new RefType( tempKlasse.get_Superclass_Name(),
              ((UsedId)tempKlasse.superclassid).vParaOrg )
                  );
Pair P2 = new Pair( P1.getTA1Copy(), P1.getTA1Copy() );
```

Das Paar `P1` enthält die aktuelle Klasse (`tempKlasse.get_classname`) aus dem Klassenvektor und ihre Superklasse (`tempKlasse.get_Superclass_Name`) jeweils mit den Originalparametern (`vParaOrg`). In [4] werden die Parameter der Superklasse in den Parametervektor der Basisklasse übernommen. Hier werden aber die originalen Parameter, wie sie vom Parser erkannt werden, benötigt. Deshalb wurden die Attribute `vParaOrg` in `Class` und `UsedID` eingeführt.
Das Paar `P2` enthält die aktuelle Klasse und nochmals die aktuelle Klasse (reflexive Eigenschaft von FC).
Nachdem der erste Teil von `makeFC` ausgeführt wurde, sieht die Menge FC für das vorhergehende Beispiel wie folgt aus:

```
Menge FC =  {
               (Vektor< A >, Vektor< A >),
               (Vektor< A >, AbstractList< A >),
               (Matrix< A >, Matrix< A >),
               (Matrix< A >, Vektor< Vektor< A > >),
              (ExMatrix< A >, ExMatrix< A >),
               (ExMatrix< A >, Matrix< A >)
            }
```

Teil 2 von `makeFC`:

Dieser Teil ist etwas komplexer. Ein Flussdiagramm soll zum Verständnis beitragen.


Teil1 von makeFC
Menge FC: vFC
Wurde (im vorigen Schleifendruchlauf) neues Paar in FC eingefügt? (am Anfang: JA)
NEIN
return vFC;
In rechten Paarseiten von vFC nach Typkonstruktoren suchen
JA
Wurden alle Elemente von FC durchlaufen?
NEIN
Erstes / nächstes Element aus vFC als Pair „casten“
Alle Parameter des rechten Typausdrucks des Paares durchlaufen, beim 1. Typkonstruktor anhalten, Nummer des Parameters, der Typkonstruktor ist, in "nSubstStelle" speichern
Wurde Typkonstruktor gefunden?
NEIN
JA
Transitivität berechnen
1. Schleife: vFC durchlaufen
2. Schleife: vFC durchlaufen
Rechte Seite des aktuellen Paares der Schleife 1 mit linker Seite des aktuellen Paares der Schleife 2 vergleichen
Sind die betrachteten Typausdrücke exakt gleich?
NEIN
Schleifen-durchläufe fortsetzen
JA
Neues Paar anlegen: Pair(PaarSchleife1.TA1, PaarSchleife2.TA2)
Angelegtes Paar in vFC einfügen, falls noch nicht vorhanden; Prüfen mit Pair.isInVector
vFC von vorne durchlaufen: Typ, der als Parameter den gefundenen Typkonstruktor enthält auf der linken Seite eines vFC-Paares suchen
Wurde solch ein Paar gefunden?
NEIN
JA
Paar kopieren
TyploseVariable an der nSubstStelle-ten Stelle der Parameterliste vom linken Typausdruck des kopierten Paares merken
TyploseVariable im linken Typausdruck und im rechten Typausdruck des kopierten Paares durch den gefundenen Typkonstruktor ersetzen bzw. substituieren mit der Funktion subst
Kopiertes Paar in vFC einfügen, falls noch nicht vorhanden; Prüfen mit Pair.isInVector

Der schwarz hinterlegte Prozess soll anzeigen, dass hier eine Einschränkung des Algorithmuses vorgenommen wurde. Denn beim 1. gefundenen Typkonstruktor werden die restlichen Parameter, die u.U. weitere Typkonstruktoren enthalten, vernachlässigt. An dieser Stelle ist die Bestimmung der Menge FC nicht vollständig. Auf Javaprogrammen, in denen mehrere Typkonstruktoren in einer Parameterliste vorkommen, funktioniert die Bildung der Menge FC nicht korrekt.
Die verwendete Funktion `subst` wird im nächsten Kapitel erläutert.

4.3.3. Die Funktion `unify`

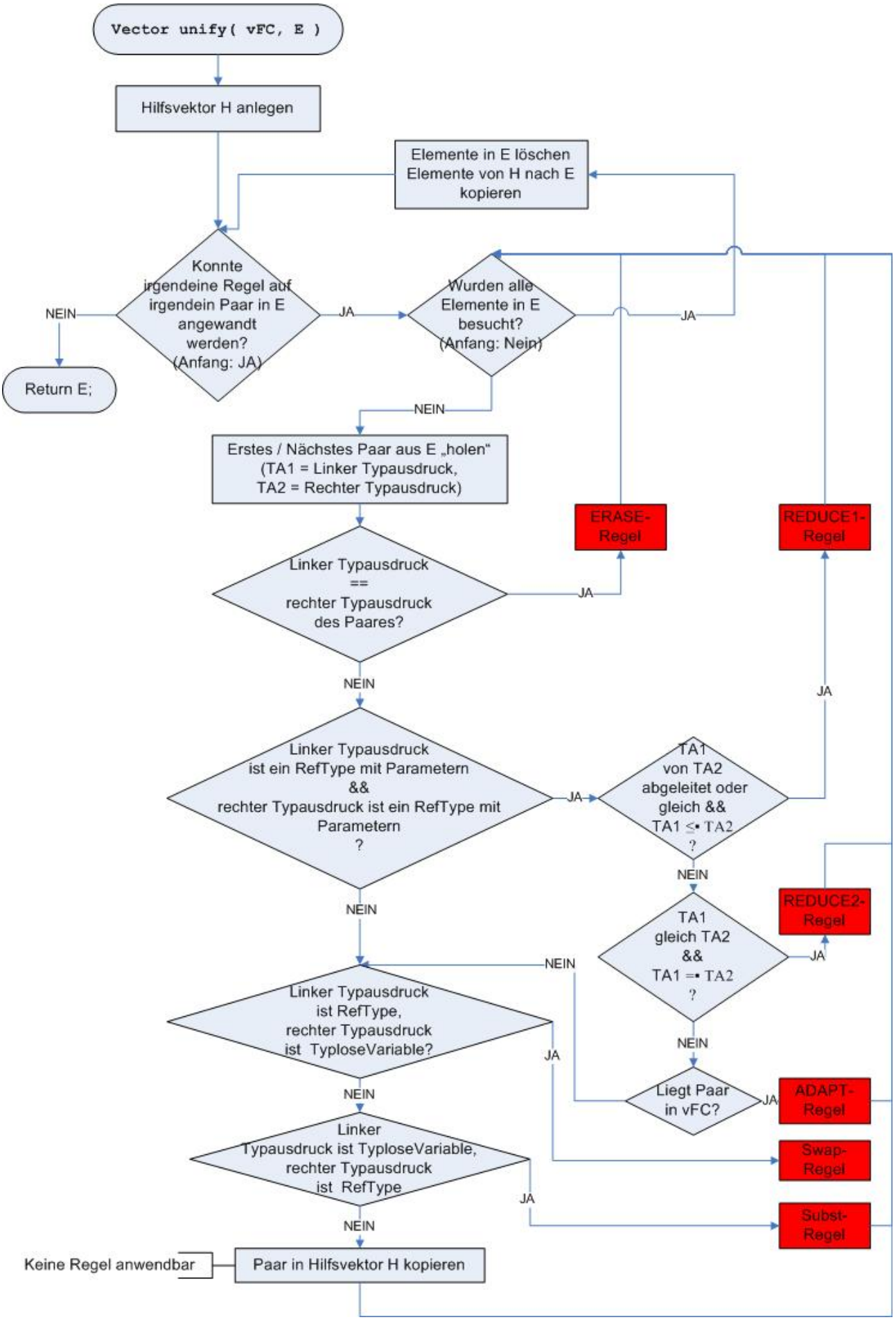
Die eigentliche Unifikation erfolgt in dieser Funktion. Als Parameter bekommt sie eine Anfangsmenge E (in der Definition mit Eq bezeichnet) und die berechnete Menge FC übergeben. Um Mengen zu speichern wird generell die Klasse `Vector` verwendet, obwohl sich vielleicht die Klasse `Set` besser eignen würde.

Zuerst wird der grobe Ablauf beschrieben. Danach werden die verwendeten Funktionen weiter im Detail beschrieben.

Beispiele:
Um den Unifikationsalgorithmus zu testen wurden Beispiele „von Hand“ aufgebaut. Das heißt die Menge E wurde manuell erstellt. Im Rahmen der Typinferenz wird die Menge E automatisch aufgebaut.
Beispiel: E = { (A, Stack<B>), (Stack<IntZahl>, B) }

```
Vector vParameter = new Vector();
vParameter.add( new TyploseVariable("B") );
RefType TA12 = new RefType( "Stack", vParameter );
Vector vParameter2 = new Vector();
vParameter2.add( new RefType("IntZahl") );
RefType TA21 = new RefType( "Stack", vParameter2 );
Pair P1 = new Pair( new TyploseVariable("A"), TA12 );
Pair P2 = new Pair( TA21, new TyploseVariable("B") );
Vector E = new Vector();
E.addElement( P1 ); E.addElement( P2 );
```

Grober Ablauf:
Die Struktur der Unifikation wird im nächsten Flussdiagramm erklärt. Darin ist zu sehen, dass es zwei verschachtelte Schleifen gibt. Die innere Schleife prüft, ob eine Regel auf ein Paar der Menge E anwendbar ist. Ist dies der Fall, wird die Regel angewandt. Die durch die Regeln erzeugten Paare bzw. die unveränderten Paare werden in den Hilfsvektor H kopiert. Nachdem alle Paare in dem Vektor E „besucht“ wurden, wird der Vektor E gelöscht und die Paare vom Vektor H nach E kopiert. Danach wird der Vektor H gelöscht. Die Menge H wird also am Schleifenende zur neuen Menge E. Die Hilfsmenge H sollte es einfach machen, die dynamisch veränderliche Menge Eq (bzw. E) zu programmieren.
Nun folgt der nächste Schleifendurchlauf der äußeren Schleife. Falls in der inneren Schleife keine Regel angewandt werden konnte (dies wird über eine boolesche Variable gesteuert), wird der Vektor E als Ergebnis zurückgegeben. Ansonsten wird die innere Schleife wieder angesprungen und versucht im Vektor E Regeln anzuwenden. Die rot hinterlegten Rechtecke stehen für Regel-Prozesse. Diese werden im Einzelnen noch näher erläutert.


Vector unify(vFC, E)
Hilfsvektor H anlegen
Elemente in E löschen
Elemente von H nach E kopieren
Konnte irgendeine Regel auf irgendein Paar in E angewandt werden? (Anfang: JA)
NEIN
JA
Return E;
Wurden alle Elemente in E besucht? (Anfang: Nein)
JA
NEIN
Erstes / Nächstes Paar aus E „holen“ (TA1 = Linker Typausdruck, TA2 = Rechter Typausdruck)
Linker Typausdruck == rechter Typausdruck des Paares?
JA
ERASE-Regel
REDUCE1-Regel
NEIN
Linker Typausdruck ist ein RefType mit Parametern && rechter Typausdruck ist ein RefType mit Parametern ?
JA
TA1 von TA2 abgeleitet oder gleich && TA1 ≤• TA2 ?
JA
NEIN
NEIN
TA1 gleich TA2 && TA1 =• TA2 ?
REDUCE2-Regel
JA
NEIN
Linker Typausdruck ist RefType, rechter Typausdruck ist TyploseVariable?
JA
NEIN
Liegt Paar in vFC?
JA
ADAPT-Regel
Linker Typausdruck ist TyploseVariable, rechter Typausdruck ist RefType
Swap-Regel
JA
NEIN
Subst-Regel
Keine Regel anwendbar
Paar in Hilfsvektor H kopieren

Erase-Regel

Indem das betroffene Paar nicht in den Hilfsvektor H kopiert wird, wird es de facto aus E gelöscht, da die Menge H im nächsten Schleifendurchlauf zur Menge E wird.

Reduce1-Regel

Laut Definition (s. Kapitel 4.2) muss die Klasse des linken Typausdrucks (TA1) von der Klasse des rechten Typausdrucks (TA2) abgeleitet oder gleich sein. Gleichheit wird über einen Stringvergleich herausgefunden, die Vererbung wird über die Funktion `isRealSubClass` ermittelt. Der Stringvergleich ist an dieser Stelle zulässig, weil für zwei verschiedene Typausdrücke zwei verschiedene Strings entstehen (ohne Beweis).
Zusätzlich muss TA1 =• TA2 gelten. Diese Bedingung wird über die boolesche Variable `Pair.bEqual` geprüft.
Weiterhin muss die Anzahl der Typparameter von TA1 und TA2 gleich sein. Zusätzlich dürfen als Parameter (in der Klassendefinition) keine Typkonstruktoren vorkommen. Bisher wird lediglich die erste Bedingung über die Vektor-Funktion `size` geprüft.
Bei der Reduce-Regel entstehen so viele Paare wie Parameter vorkommen. Diese Paare werden in einer Schleife über die Parameter mit folgender Zeile erzeugt:

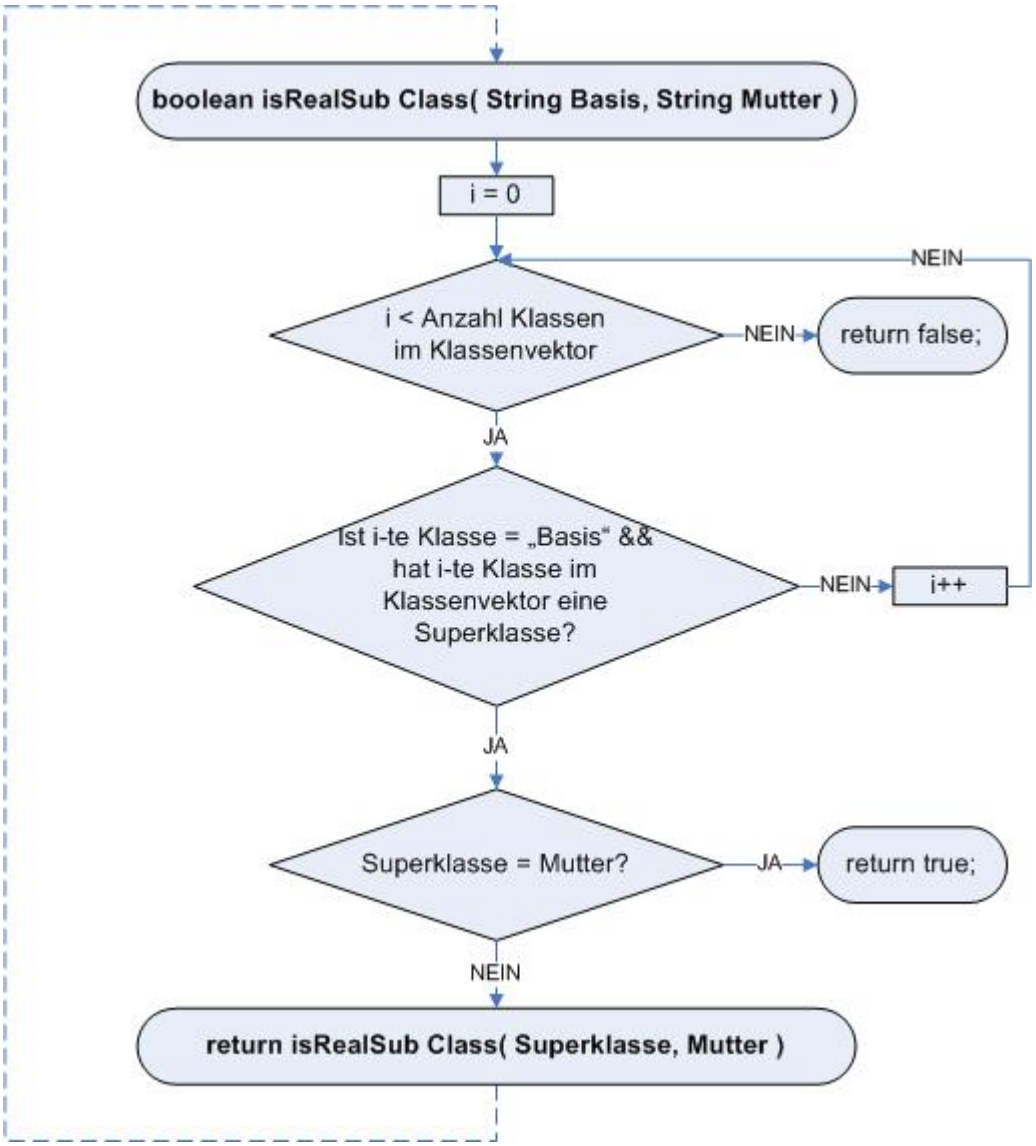
```
Pair P2 = new Pair(
  (Type)L1.elementAt(pi(k,TA1.getTypeName(),TA2.getTypeName())),
  (Type)L2.elementAt(k), true ); // k ist "Laufvariable"
```

Ein neues Paar besteht jeweils aus dem `pi(k)`-ten Parameter von TA1 und des k-ten Parameters von TA2. Die Funktion `pi` bildet die Permutation der Parameter, die sich durch die Klassendefinition der Klassen von TA1 und TA2 ergeben.

Wichtig ist hierbei noch, dass für das neue Paar die Variable `Pair.bEqual` auf `true` gesetzt wird (s. `true` im Beispiel). Dies bedeutet für das Verhältnis der beiden Ausdrücke des neuen Paares: P2.TA1 =• P2.TA2. Dieses Paar kann somit auf keinen Fall mehr durch die Reduce1-Regel behandelt werden.
Die neu erzeugten Paar werden im Hilfsvektor H gespeichert.

Die Funktion `isRealSubClass`

Diese Funktion bestimmt, ob die Klasse `Basis` (1. übergebener Parameter) von der Klasse `Mutter` (2. übergebener Parameter) abgeleitet ist. Ist dies der Fall, wird `true` zurückgegeben. Die Bestimmung erfolgt rekursiv, da Transitivitäten auch berücksichtigt werden müssen.



Die Funktion `pi`

Die Funktion `pi` berechnet die Permutation, die für die Reduce1- und Adapt-Regel benötigt wird und durch die Klassendefinition festgelegt ist.
Dazu ein Beispiel:

Klassendefinition:

```
class X<a, b, c> extends class Y<c, b, a>
```

Permutation:

p = { (1, 3), (2, 2), (3, 1) }

Erklärung: p (1) = 3 → Das “a” steht einmal an Position 1 und einmal an Position 3.

Die Funktion `pi` ist wie folgt definiert:

```
int pi( int n, String C, String D ) throws SCException
```

Die Strings `C` und `D` stellen Klassen dar aus denen die Permutation an der Stelle `n` berechnet werden soll.
Zuerst werden die Klassen im Klassenvektor gesucht. Dann wird der Parameter / die Typvariable an der Stelle `n` von der Klasse `D` ermittelt. In einer nachfolgenden Schleife über die Parameter der Klasse `C` wird die Position der gesuchten Typvariable ermittelt und zurückgegeben. Im Fehlerfall wird eine „Exception“ geworfen.

Reduce2-Regel

Bei dieser Regel muss die Klasse von TA1 gleich sein wie die Klasse von TA2. Die Gleichheit wird mit der Funktion `String.equals` ermittelt.
Zusätzlich muss TA1 =• TA2 gelten (im Gegensatz zur Reduce1 mit =•). Diese Bedingung wird über die boolesche Variable `Pair.bEqual` geprüft.
Weiterhin muss auch hier die Anzahl der Typparameter von TA1 und TA2 gleich sein. Die Gleichheit wird über die Funktion `Vector.size` ermittelt.
Die Erzeugung der Paare erfolgt gleich wie bei der Reduce1-Regel.

Swap-Regel

Diese Regel verlangt, dass TA1 keine Typvariable ist (also ein RefType) und TA2 eine Typvariable ist. Die Prüfung erfolgt mit dem `instanceof`-Operator.
Die Swap-Regel tauscht einfach die beiden Typausdrücke eines Paares. Folgende Zeile nimmt das Taschen vor und speichert das neue Paar in H.

```
H.addElement( new Pair( P.TA2, P.TA1, true ) );
```

Adapt-Regel

Um die Adapt-Regel für ein Paar P = (TA1, TA2) anwenden zu können, muss es in der Menge FC (also im Vektor vFC) liegen. Die Funktion `isInFC` findet heraus, ob ein Paar in FC liegt. Dabei müssen die einzelnen Werte für Typvariablen in dem Paar ignoriert werden.
Beispiel:

```
vFC = { ( AA<a, b> =? CC< DD<b, a> > ), … }
P = ( AA<Integer, b>, CC< DD<Char, a>> )
```

Das Paar P liegt in vFC, denn die Substitutionen für die Typvariablen spielen keine Rolle.

Liegt das Paar in FC kann die Adapt-Regel angewandt werden.
Mit einer Schleife über die Typvariablen / Parameter der Klasse von TA1 (hier AA) wird pro Typvariable mit der Funktion `adapt`, entsprechend der Definition der Regel, ein neues Paar gebildet. Dieser Funktion werden folgende Parameter übergeben:

- die bzgl. der Schleife aktuelle Typvariable (z.B. b),
- die entsprechende Substitution des Parametervektors von TA2 (hier auch b) [:=T],
- der Parametervektor (:=vPFC) des rechten Typausdrucks des korrespondierenden Paares in FC (hier `DD<b, a>`) und
- der Parametervektor (:=vP2) von TA2, (hier `DD<Char, a>`).

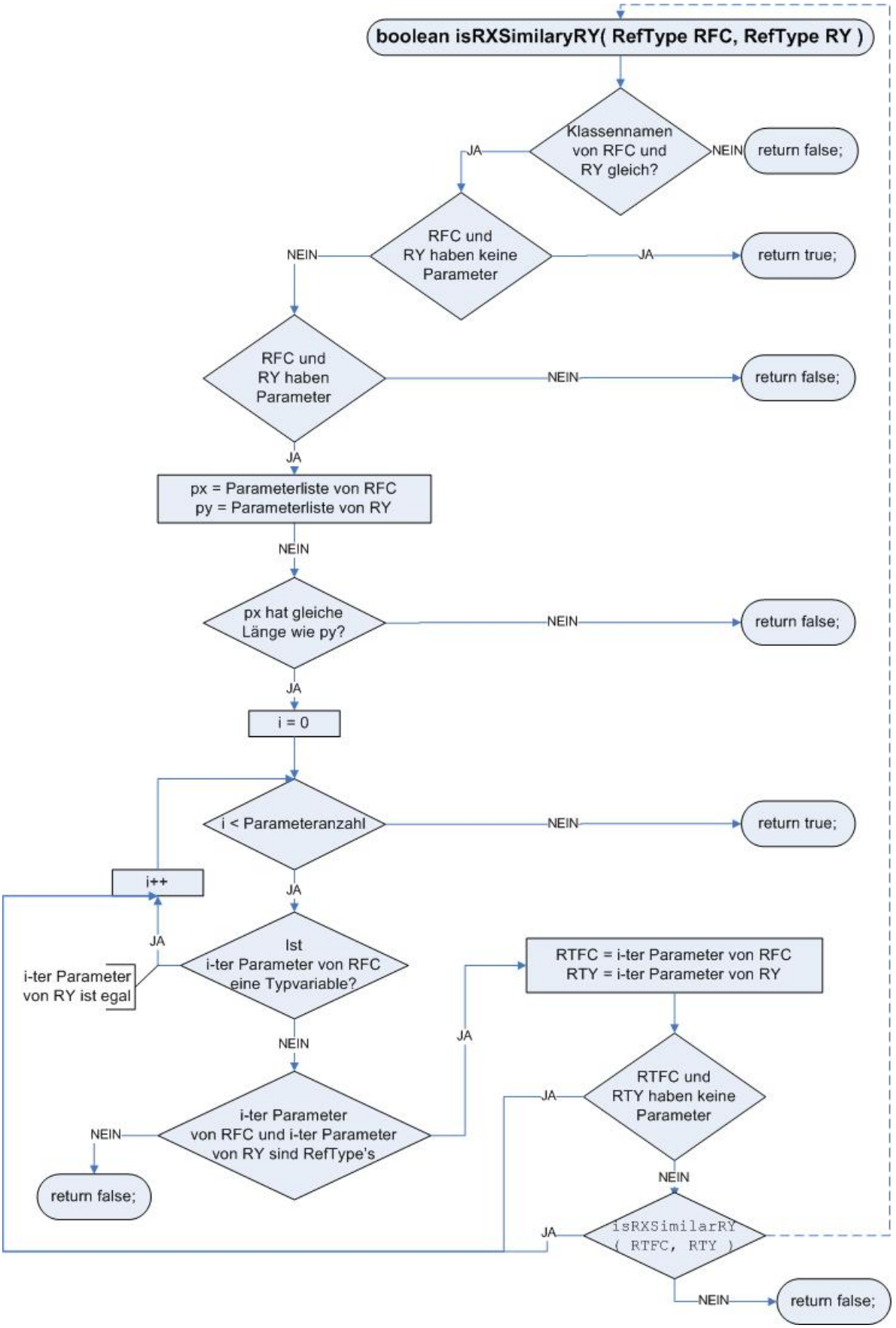
In der Funktion `adapt` wird dann die Position der Typvariablen in „vPFC“ gesucht (hier Position 1). Ein rekursiver Aufruf ist dann notwendig, wenn die Parameter einen Baum bilden.
Das neue Paar wird aus „T“ (= übergebener Parameter, s.o.) als linkem Typausdruck und der Substitution der Typvariablen an der gefundenen Position in „vP2“ als rechter Typausdruck verwendet. Hier: P_{Neu} = (b, Char)
Die Funktionsweise von `adapt` wird in einem Flussdiagramm näher erläutert.

Die Funktion `isInFC`:

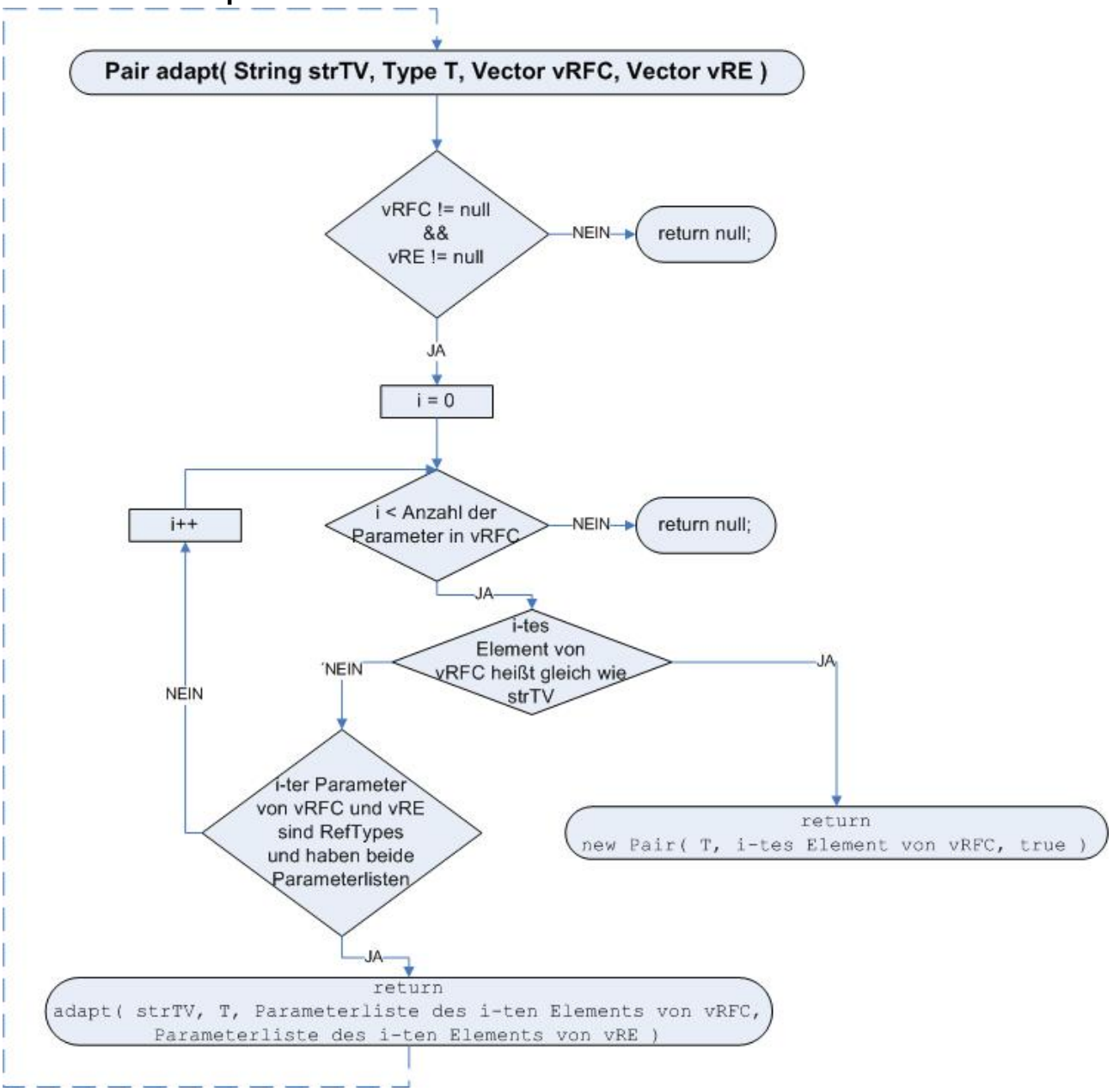
Diese Funktion prüft, ob ein übergebenes Paar P mit TA1 und TA2 in der Menge vFC liegt. Eine Schleife über den Vektor vFC überprüft jedes Paar in FC, ob es von der Struktur her in P liegt (Dieser Vorgang wird auch als „Matchen“ bezeichnet). Zuerst wird mit der Funktion `isRXSimilarRY` überprüft ob TA1 ähnlich (abgesehen von den Substitutionen) ist wie der linke Typausdruck des betrachteten Paares aus FC. Danach wird geprüft, ob TA2 mit dem rechten Typausdruck des betrachteten Paares aus FC

übereinstimmt/“matched“. Stimmen beide Seiten überein, wird das Paar aus vFC zurückgegeben. Ansonsten wird `null` zurückgegeben.

Die Funktion `isRXSimlarRY`:



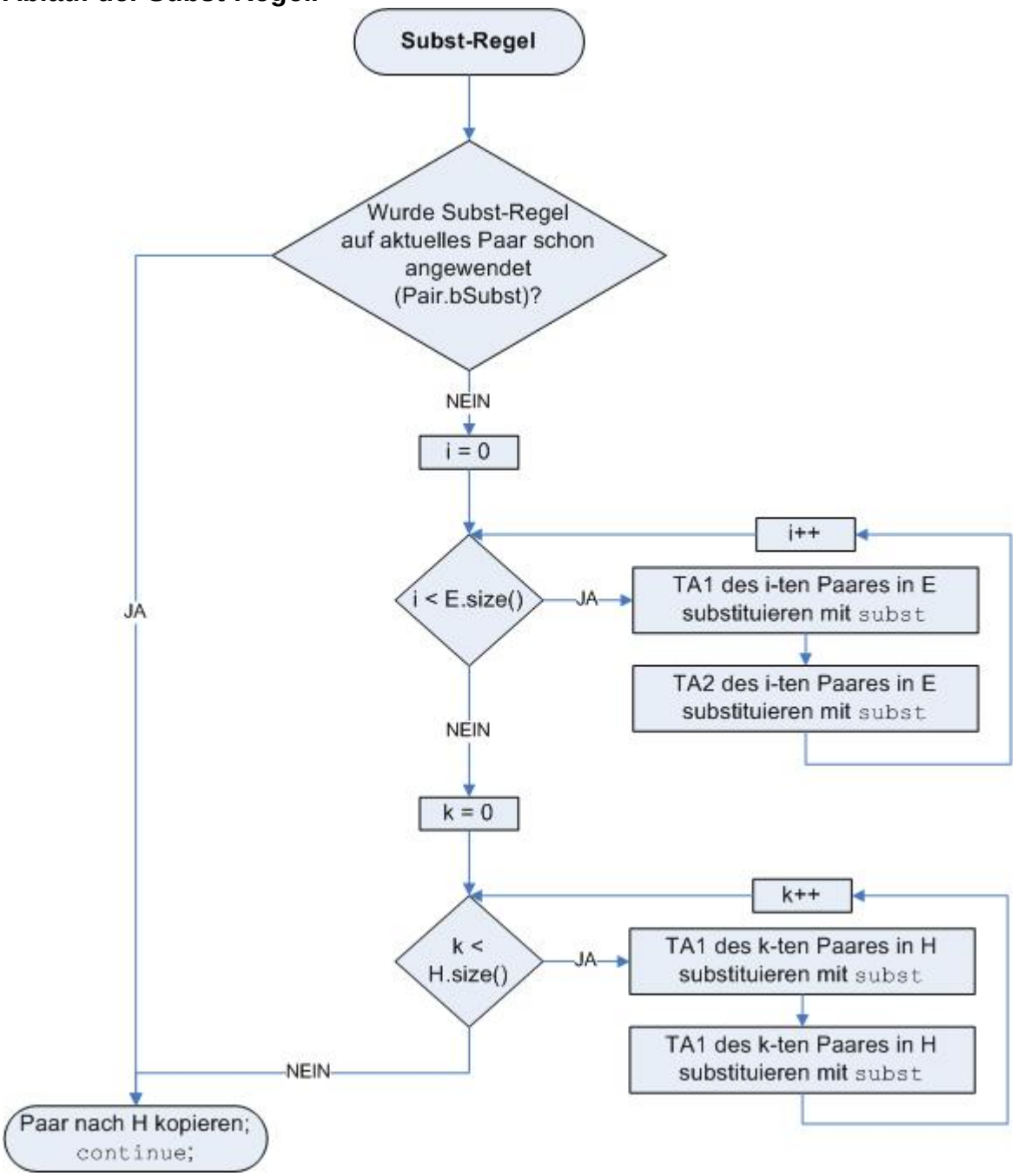
Die Funktion `adapt`:



Subst-Regel

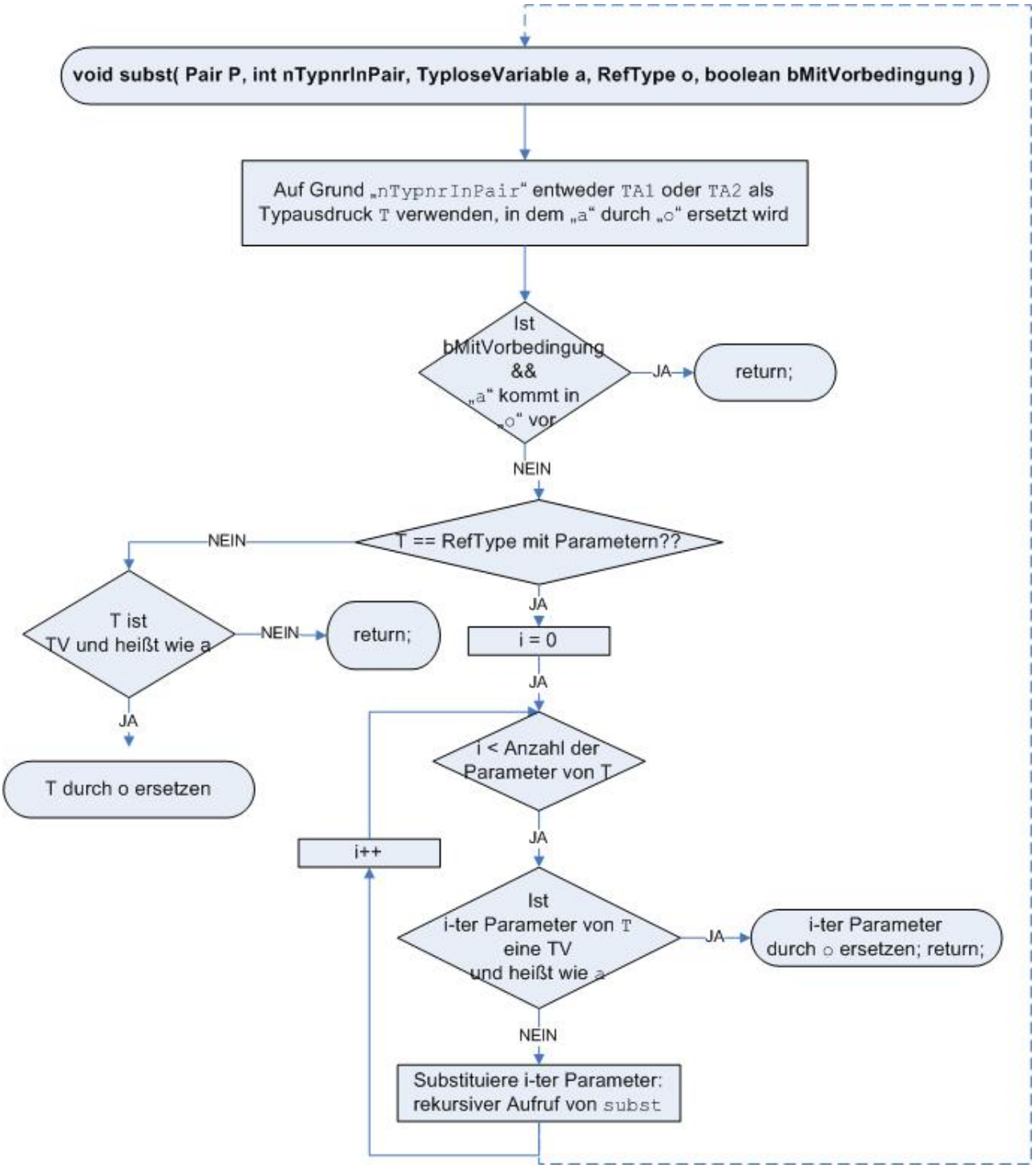
Die Subst-Regel kommt zum Tragen, wenn TA1 eine Typvariable ist und TA2 diese Typvariable nicht enthält. Zusätzlich muss `Pair.bEqual` wahr sein. Die Typvariable TA1 wird in allen Paaren in E und H, in denen diese vorkommt durch TA2 ersetzt.
Trifft für ein Paar die Substitutionsregel zu und wird diese durchgeführt, so wird die Variable `Pair.bSubst` auf `true` gesetzt. Ist die Variable gesetzt, so wird die Substitutionsregel für dieses Paar nicht mehr angewandt. Ohne diese Hilfsvariable, die per Default auf `false` gesetzt ist, würde die Subst-Regel für ein Paar unendlich oft „angesprungen“ werden.

Ablauf der Subst-Regel:



Die Substitution erfolgt mit der Funktion `subst`. Diese muss prinzipiell einen Parameterbaum durchlaufen und kann sich rekursiv aufrufen (s. Flussdiagramm unten).

Die Funktion `subst`:



Die Funktion `isTVinRefType`:

Um zu entscheiden, ob eine Typvariable in einem Typausdruck vorkommt, wird die Funktion `isTVinRefType` verwendet.
Zuerst wird geprüft, ob der übergebene RefType überhaupt Parameter besitzt. Ist dies der Fall, werden die Parameter in einer Schleife verglichen. Ansonsten wird `false` zurückgegeben. Ist ein Parameter eine TyploseVariable, die den gleichen Namen hat wie die übergebene TyploseVariable, wird `true` zurückgegeben. Handelt es sich bei dem Parameter wieder um einen RefType wird rekursiv geprüft, ob in diesem RefType die gesuchte Typvariable vorkommt. Falls am Schleifenende keine Treffer gefunden wurde, wird `false` zurückgegeben.

4.3.4. „Solved form“

Nach der Berechnung der Menge `Eq` wird geprüft, ob sie in „solved form“ (s. Kapitel 4.2) vorliegt. Die Funktion `hasSolvedForm` nimmt diese Prüfung vor.
In einer Schleife über den Vektor `Eq` wird für jedes Element untersucht, ob es folgende Bedingungen erfüllt:

- Element ist vom Typ `Pair`.
- Pair.TA1 ist vom Typ TyploseVariable.
- Pair.TA2 ist vom Typ RefType.
- Pair.TA2 enthält keine der in Pair.TA1 vorkommenden `TyploseVariablen`. Die Prüfung erfolgt mit der Funktion isTVinRefType (s. Kapitel 4.3.3).

5. Schlussbetrachtung

Diese Arbeit hat den ersten Teil des Typinferenz-Algorithmuses implementiert: die Unifikation. In einem zweiten Schritt müssen nun die Typinferenzregeln noch implementiert werden.
Bei der Unifikation wurden im Wesentlichen zwei Einschränkungen gemacht: Die Ergebnismenge der Unifikation wurde auf eine Teilmenge gekürzt und die Berechnung der Menge FC ist nicht korrekt, wenn in einer Parameterliste mehrere Typkonstruktoren vorkommen.
Der verwendete Compiler wurde im Rahmen dieser Arbeit an manchen Stellen von Fehlern befreit. Er ist aber immer noch nicht vollständig fehlerfrei.

6. Quellenangabe

[0]
Martin Plümicke. Generic Java Type Unification

[1]
Martin Plümicke. Type Inference in Generic Java

[2]
http://www.pizzacompiler.sourceforge.net
Gesichtet am: 13.04.2004

[3]
Gilad Bracha. Generics in the Java Programming Language

[4]
Felix Reichenbach. Studienarbeit. 2003

[5]
http://www.pms.informatik.uni-muenchen.de/mitarbeiter/bry/
Gesichtet am: 04.06.2004

[6]
Bauer, Höller. Übersetzung objektorientierter Programmiersprachen

[7]
Markus Haas. Studienarbeit. Juni 2004

[8]
Robin Milner. The definition of Standard ML (Revised). MIT Press, Cambridge, Mass., 1997.

7. Anhang

Der Studienarbeit liegt eine CD mit folgendem Inhalt bei:

- Projektdateien des Compilers
- Studienarbeit als PDF-Dokument
- Studienarbeit als Word-Dokument