STUDIENARBEIT

Berufsakademie Stuttgart - Außenstelle Horb am Neckar
Staatliche Studienakademie

Fachrichtung Informationstechnik

Thema

Typinferenz in Java

16. Juni 2005

Eingereicht von:

Jörg Bäuerle
Höllweg 105
72270 Baiersbronn

Ausbildungsbetrieb:

ARBURG GmbH + Co KG
Arthur-Hehl-Strasse
72286 Loßburg

Betreuer:

Dr. Martin Plümicke

Erklärung

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

Horb, den 16. Juni 2005

Jörg Bäuerle
TIT2002, BA Stuttgart - Außenstelle Horb a. N.

Typinferenz in Java

„Kein Compilerbau - keine Probleme!“
von Dr. Olaf Herden

Zusammenfassung

Sowohl imperative Programmiersprachen wie Pascal oder C als auch objektorientierte Sprachen wie Java oder C++ verlangen für jede Variable und jede Funktion eines Programms eine explizite Typdeklaration. Dem gegenüber stehen funktionale Sprachen wie z.B. SML, die ein Typinferenzsystem besitzen, das für gegebene Deklarationen die zugehörigen Typen berechnet.

Die vorliegende Studienarbeit befasst sich mit dem Implementieren und Integrieren eines Typinferenzalgorithmus und dessen Unteralgorithmen in einen bestehenden Java Compiler. Dieser Algorithmus soll beim Kompilieren eines Java-Programms die Typen von Variablen und Funktionen rekonstruieren, so dass der Programmierer nicht mehr gezwungen ist, im Quellcode alle Typen explizit anzugeben.

Abstract

In well-known imperative programming languages like Pascal or C as well as in object-oriented languages like Java or C++ an explicite type declaration for variables and functions is mandantory. In contrast there are functional languages like SML which have a type inference system determining types for given declarations.

The present student research project deals with the implementation and integration of a type inference algorithm and its subalgorithms into an existing Java compiler. This algorithm is to reconstruct a Java program's types during the compile process, so that the programmer is not obliged to declare all of the types explicitly.

Vorwort

Mein besonderer Dank gilt Dr. Martin Plümicke für seine Geduld beim Erklären seiner Algorithmen und beim Diskutieren kniffliger Implementierungsfragen. Die Studienarbeit war genau so, wie ich sie mir erhofft hatte. Sie enthielt äußerst anspruchsvolle und fordernde Elemente der theoretischen Informatik, war aber dennoch praxisbezogen.

Ebenso danke ich Dr. Olaf Herden für seine humorvolle Art, die doch manchmal recht mühselige Arbeit in einem anderen Licht erscheinen zu lassen: „Compiler baut man nicht, man benutzt sie!“

Weiter danke ich Thomas Ott für die Kurzeinführung in seine Studienarbeit und das bestehende Projekt.

Meinem Studienarbeitskollegen Markus Melzer danke ich für seine ganz besondere Begabung, mich in stressreichen Zeiten zum Lachen zu bringen. Ich hätte mir keinen anderen Kommilitonen für die Partnerstudienarbeit gewünscht.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 Motivation . . . 7
- 1.2 Aufgabenbeschreibung . . . 7
- 1.3 Projektgeschichte . . . 7
- 1.4 Projektplanung . . . 8

2 Theoretische Grundlagen **9**
- 2.1 Java . . . 9
  - 2.1.1 Die Programmiersprache Java . . . 9
  - 2.1.2 Die Java Virtual Machine . . . 9
  - 2.1.3 Die Java-Plattform . . . 10
- 2.2 Compiler . . . 10
  - 2.2.1 Analysephase . . . 11
  - 2.2.2 Synthesephase . . . 11
- 2.3 Generische Typen . . . 12
  - 2.3.1 Kopierende Sicht und C++ Templates . . . 12
  - 2.3.2 Generische Sicht und Java Generics . . . 15
  - 2.3.3 Typterme in Generic Java . . . 17
- 2.4 Typinferenz . . . 18

3 Anpassung des bestehenden Compilers **21**
- 3.1 Analyse des Build Process . . . 21
- 3.2 Neustrukturierung des Softwareprojektes . . . 22
- 3.3 Änderungen am Quellcode . . . 23
  - 3.3.1 TyploseVariable vs. GenericTypeVar . . . 23
  - 3.3.2 Einführung von BaseTypes . . . 24
  - 3.3.3 Korrekturen bei der Verwendung des Typs Void . . . 24
  - 3.3.4 Einführung von Zeilennummern . . . 24

4 Der Typinferenzalgorithmus **25**
- 4.1 Allgemeine Beschreibung . . . 26
- 4.2 Grundlegende Datenstrukturen . . . 26
  - 4.2.1 Die Klasse CTypeAssumption . . . 26
  - 4.2.2 Die Klasse CTypeAssumptionKey . . . 27

4.2.3 Die Klasse CSubstitution . . . 28
4.2.4 Die Klasse Pair . . . 28
4.2.5 Implementierung von Mengen . . . 28
4.2.6 Die Klasse CIntersectionType . . . 29
4.2.7 Die Ergebnisdatenstruktur CTypeReconstructionResult . . . 29
4.3 Kopplung von Algorithmus und Syntaxbaum . . . 30
4.4 Substitutionsbasiertes Konzept . . . 32
4.5 Die Klasse GenericTypeVar . . . 33
4.6 Das IDE-Interface . . . 34
4.7 Exception Handling . . . 35
4.8 Unteralgorithmen . . . 35
4.8.1 Die Einstiegsmethode typeReconstruction() . . . 35
4.8.2 Der Unteralgorithmus TRProg . . . 35
4.8.3 Der Unteralgorithmus TRStart . . . 36
4.8.4 Der Unteralgorithmus TRNextMeth . . . 36
4.8.5 Der Unteralgorithmus TRStatements . . . 37
4.8.6 Die Unteralgorithmen TRStatement und TRExp . . . 37
4.8.7 Die Unteralgorithmen TRTuple und TRMultiply . . . 38
4.9 Zusammenfassung des Programmablaufs . . . 38
4.10 Abweichungen von [Plü04a] . . . 39
4.10.1 Lokale Variablen . . . 39
4.10.2 TRStatement für If-Anweisung . . . 40
4.10.3 TRExp für Operatoren . . . 40

5 Schlussbetrachtungen **41**
5.1 Zusammenfassung . . . 41
5.2 Ausblick . . . 41

A Einrichten der Entwicklungsumgebung **43**
A.1 Installation von Eclipse . . . 43
A.2 Einrichten von CVS . . . 43
A.3 Auschecken und Konfigurieren des Projektes . . . 45

B Usecase-Programm **47**

C Plümickesche Algorithmen **48**

Abkürzungsverzeichnis **66**

Literaturzverzeichnis **67**

Kapitel 1

Einleitung

1.1 Motivation

In der Programmiersprache Java ist es nötig, die Typen von Variablen und Methoden alle explizit anzugeben. Diese Typangaben können vorallem bei komplexeren Ausdrücken für den Programmierer schnell lästig und zum Teil recht unübersichtlich und verwirrend werden. Mit der Einführung generischer Typen im JDK 1.5 von Sun wurden die anzugebenden Typterme noch komplexer.
Hier wäre es für einen Java-Programmierer hilfreich, wenn er das Einfügen der Typinformationen dem Compiler selbst überlassen könnte.

1.2 Aufgabenbeschreibung

Aufgabe der vorliegenden Studienarbeit ist das Einführen eines Typinferenzsystems in die Programmiersprache Java, das die explizite Typangabe bei Variablen- und Methodendeklarationen überflüssig macht. Hierzu soll der in [Plü04a] spezifizierte Typinferenzalgorithmus, auch als Typrekonstruktionsalgorithmus[1] bezeichnet, und dessen Unteralgorithmen vollständig implementiert und in einen bereits bestehenden Java Compiler integriert werden.
Darüberhinaus soll eine Schnittstelle zur Studienarbeit „Integration der Java-Typinferenz in eine Programmierumgebung“ geschaffen werden, die sich mit der Integration des Java Compilers in die IDE „Eclipse“ befasst.

1.3 Projektgeschichte

Grundlage für dieses Projekt ist ein in Java implementierter Java Compiler, der von Studenten des Jahrganges TIT 2000 im Rahmen der Vorlesungen „Informatik 4“ und „Software Engineering“ entwickelt wurde. Dieser Compiler wurde in den Vorgängerstudienarbeiten [Rei03] und [Haa04] um die Unterstützung von generischen Typen erweitert. Des weiteren wurde der

[1] Wird im Folgenden als „TRA“ abgekürzt.

Unifikationsalgorithmus, der vom zu implementierenden TRA verwendet wird, mit einigen Einschränkungen bereits in [Ott04] implementiert.

1.4 Projektplanung

Die Durchführung des Projektes gliedert sich wie folgt in zwei Theoriephasen und eine dazwischen liegende Praxisphase.

Theoriephase 5. Semester:	Projektanalyse und Erarbeitung theoretischer Grundlagen
Praxisphase 5. Semester:	Implementierung der Datenstrukturen und Anpassung des abstrakten Syntaxbaums
Theoriephase 6. Semester:	Implementierung des TRA und dessen Unteralgorithmen

Kapitel 2

Theoretische Grundlagen

2.1 Java

Die Urversion von *Java*, auch Oak genannt, wurde von 1991 bis 1992 unter dem Projektnamen *The Green Project* im Auftrag des amerikanischen Computerherstellers Sun Microsystems als geräteunabhängige Entwicklungsplattform für Geräte aus dem Unterhaltungsbereich entwickelt. In den darauf folgenden Jahren nahm Java-Technologie mehr und mehr Einzug in viele Bereiche der IT-, Elektronik- und Unterhaltungsbrache. Heutzutage ist der Begriff Java in aller Munde, wobei es jedoch äußerst wichtig ist, zwischen den einzelnen Komponenten, die diese Technologie ausmachen, genau zu unterscheiden. Die drei großen Hauptbestandteile von Java sind die Programmiersprache Java, die Java Virtual Machine (JVM) und die Plattform Java (vgl. im Folgenden [Fla00] und [Wik05]).

2.1.1 Die Programmiersprache Java

Die Programmiersprache Java ist die Sprache, in der Java-Applikationen geschrieben werden. Sie besitzt eine C-ähnliche Syntax und ist objektorientiert, wobei jedoch einige Merkmale, die andere objektorientierte 3GL-Sprachen enthalten, wie z.B. Zeiger oder Mehrfachvererbung von den Sprach-Designern nicht integriert wurden. Der objektorientierte Ansatz ermöglicht es, große Softwareprojekte einfacher managen zu können und einen hohen Grad der Wiederverwendung von Softwaremodulen zu erreichen.

2.1.2 Die Java Virtual Machine

Unter der Java Virtual Machine versteht man die Laufzeitumgebung (Runtime Environment), die benötigt wird, um Java-Programme auszuführen. Der Java-Quellcode wird vom Java-Compiler nicht direkt in ausführbare Maschinensprache übersetzt, sondern in einen maschinen- und plattformunabhängigen *Bytecode* umgewandelt. Dieser Bytecode wird zur Laufzeit von der JVM interpretiert und in native Maschinensprache konvertiert, wobei man in diesem Zusammenhang auch von *Just-In-Time-Kompilierung* (JIT) spricht.
Somit sind Java-Programme in dem Sinne plattformunabhängig und portabel, dass sie unter jeder Plattform lauffähig sind, für die auch eine Implementierung der JVM verfügbar ist. So

laufen Java-Programme ebenso auf PCs unter Windows oder Unix wie auch auf PDAs oder Handys. Der wohl verbreitetste Java-Interpreter ist die JVM-Implementierung von Sun, die aktuell in der Version 1.5 erhältlich ist.[1] Des weiteren gibt es mittlerweile auch schon Laufzeitumgebungen auf Hardwarebasis in Form von Prozessoren, deren native Maschinensprache der Java-Bytecode selbst ist.

Obwohl Interpreter normalerweise nicht gerade besonders leistungsstark sind, ist die JVM auf Grund ständiger Weiterentwicklung in der aktuellen Version bemerkenswert schnell. Vor allem durch die Just-In-Time-Kompilierung wird die Ausführungsgeschwindigkeit von wiederholt ausgeführtem Code deutlich erhöht, wobei die Hotspot-Technologie von Sun eine besonders gute Implementierung von JIT darstellt.

2.1.3 Die Java-Plattform

Für einen Programmierer wird die Java-Plattform definiert durch die *Java-API*, die er beim Schreiben von Java-Programmen verwenden kann. Eine API (Application Programming Interface) ist normalerweise durch das Betriebssystem des Zielrechners vorgegeben und stellt die Schnittstelle dar, die dieses Betriebssystem anderen Programmen zur Verfügung stellt. Eine API stellt Routinen, Protokolle, Bibliotheken und Tools für das Erstellen von Software dar und erleichtert in der Regel das Programmieren durch die Bereitstellung von Software-Modulen.

Die Java-Plattform verkörpert also in gewissem Sinne selbst eine Art Betriebssystem, das von dem jeweiligen darunter liegenden System abstrahiert und eine eigene Schnittstelle, die Java-API, besitzt [Plü04b]. Die Java-API selbst ist in sogenannten *Packages* strukturiert, die Java-Klassen für Bereiche wie Input/Output, Netzwerk, Grafik, Sicherheit, usw. bereitstellen.

2.2 Compiler

In [uRH98] wird ein *Compiler* als ein Computerprogramm definiert, das ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache umwandelt. Dabei ist das Quellprogramm meist in einer höheren Programmiersprache geschrieben und wird in eine maschinennahe Sprache wie Assembler, Bytecode oder Maschinensprache übersetzt. Darüberhinaus hat der Compiler die Aufgabe im Quellprogramm benutzte Bezeichner und deren Informationen zu speichern (*Symboltabellenverwaltung*), sowie etwaige im Quellprogramm vorhandene Fehler zu erkennen und zu melden (*Fehlerbehandlung*). Der gesamte Übersetzungsvorgang wird als *Kompilieren* bezeichnet.
Der Kompiliervorgang besteht nach [AVA99] und [uRH98]aus zwei Phasen: *Analyse* und *Synthese*, die wiederum selbst in mehrere Teile untergliedert werden.

[1] Ebenfalls bekannt als *Java 5.0 Tiger*

2.2.1 Analysephase

In der Analysephase wird das flache, serialisierte Quellprogramm analysiert und in eine hierarchische Struktur, den sogenannten *Syntaxbaum*, umgesetzt [AVA99, Wik05]. Die Analysephase, die auch „Frontend“ genannt wird, besteht aus drei Teilen: Lexikalische Analyse, Syntaktische Analyse und Semantische Analyse.

Bei der *lexikalischen Analyse* zerlegt ein sogenannter *Scanner*, auch *Lexer* genannt, den eingelesenen Quelltext in logisch zusammengehörige Einheiten, die sogenannten *Tokens*. Der Scanner, der als endlicher Automat realisiert wird, zerteilt die Eingabe nach den Regeln einer regulären Grammatik und ist meist Teil eines Parsers, an den er die Tokens weiter gibt.

Nach der lexikalischen Analyse überprüft die *syntaktische Analyse* mit Hilfe eines *Parsers*, ob der eingelesene Quellcode formal richtig ist, d.h. der Syntax bzw. Grammatik der Quellsprache entspricht. Der Parser, der meist als Kellerautomat implementiert ist, arbeitet auf den Tokens, die ihm von einem Scanner geliefert werden. Die Tokens dienen dem Parser als atomare Eingabezeichen und werden von ihm zu einem *Syntaxbaum*, auch Ableitungsbaum oder Parse-Baum genannt, zusammengebaut.

Während der *semantischen Analyse* wird der nach der lexikalischen Analyse vorhandene Syntaxbaum dann auf semantische Fehler überprüft. Zwei wesentliche Elemente der semantischen Analyse sind Typüberprüfungen und das Sammeln von Typinformationen für die anschließende Phase der Code-Generierung. So werden beispielsweise Variablendeklarationen, Zuweisungen und Operationen auf Typkorrektheit überprüft.

Nach erfolgreichem Abschluß der Analysephase liegt also ein maschinenunabhängiger Syntaxbaum vor, der ein fehlerfreies Programm repräsentiert. Das heißt mit anderen Worten, dass das Programm Bestandteil der von der angegebenen Grammatik erzeugten Sprache ist.

2.2.2 Synthesephase

In der Snythesephase des Kompiliervorgangs wird aus dem nach Abschluss der Analysephase bestehenden Syntaxbaum Programmcode der Zielsprache erzeugt. Die Synthesephase kann wiederum in mehrere Teilphasen untergliedert werden.

Manche Compiler besitzen eine *Zwischencodeerzeugung*, welche aus dem abstrakten Syntaxbaum eine Zwischendarstellung des Quellprogramms für eine abstrakte Maschine erstellt [AVA99]. Dieser Zwischencode ist bereits relativ maschinennah und leicht ins Zielprogramm zu übersetzen. Eine Zwischencodeerzeugung bietet sich besonders bei Compilern an, die mehrere verschiedene Zielplattformen unterstützen.

Während der Phase der *Codeoptimierung* wird versucht den Zwischencode derart zu verbessern, dass die Laufzeit und der Speicherbedarf des Zielprogramms verkleinert wird. Die Codeoptimierung ist stark abhängig von der Hardware der Zielplattform, d.h. vom Prozessor,

auf dem das Zielprogramm später ablaufen wird.

Die letzte Compilerphase ist für gewöhnlich die *Codegenerierung*, in der der endgültige Programmcode in der Zielsprache erzeugt wird. Ist die Zielsprache die Maschinensprache, so kann das Ergebnis der Codegenerierung direkt ein ausführbares Programm sein. Im Falle des Java-Compilers wird in dieser Compilerphase Bytecode generiert.

Manche Compiler besitzen laut [uRH98] nach der Codegenerierung noch eine zweite Codeoptimierungsphase. Hierbei ist die erste Codeoptimierung nach Erzeugung des Zwischencodes im allgemeinen zielmaschinenunabhängig, während die finale Codeoptimierung den generierten Zielcode für eine spezielle Zielarchitektur, wie z.B. einen Parallelrechner mit mehreren Prozessoren, optimiert.

2.3 Generische Typen

Eines der wichtigsten Programmierparadigmen vor allem in der objektorientierten Programmierung ist die generische Programmierung, bei der die Algorithmen so weit wie möglich von den Datenstrukturen, mit denen diese arbeiten, getrennt werden [Wik05]. Dies ermöglicht es, dass derselbe Algorithmus nicht für jeden einzelnen Datentyp neu implementiert werden muss, sondern nur bestimmte Anforderungen an die jeweiligen Typen stellt. Solch ein Datentyp wird als *Generischer Datentyp* bezeichnet.
In [uRH98] wird ein generischer Datentyp als ein parametrisierter Datentyp, der mit verschiedenen Typen instanziert werden kann, beschrieben. D.h., die Implementierung ist von der Programmierersicht aus für alle Instanzen dieselbe. Eine generische Klasse ist demzufolge also eine Art Prototyp für eine Klasse, eine Meta-Klasse, mit formalen Parametern. Durch Instanzierung einer generischen Klasse erhält man eine neue Klassendefinition. Die Instanzierung erfolgt durch Angabe von aktuellen Parametern für die formalen Parameter, wobei die aktuellen Parameter die formalen Parameterrestriktionen erfüllen müssen.

Der Wunsch nach generischen Typen liegt in allen Programmiersprachen meist in der Implementierung von sogenannten Collection- oder Containerklassen. Stellvertretend hierfür wird zu Anschauungszwecken im Folgenden die Implementierung einer Container-Klasse `DataPair`, die zwei Objekte eines beliebigen Typs aufnehmen soll, betrachtet.
Bei der Realisierung generischer Datentypen gibt es nach [uRH98] zwei Varianten: die *Kopierende Sicht* und die echte *Generische Sicht*.

2.3.1 Kopierende Sicht und C++ Templates

Bei der Kopiervariante wird die generische Definition als eine Art Schablone oder Muster aufgefaßt. Bei Instantiierung der generischen Klasse wird diese kopiert und alle formalen durch aktuelle Parameter ersetzt. Dabei erzeugt der Compiler eine jeweils separate Repräsentation für jede einzelne Instanziierung eines parametrisierten Typs bzw. einer parametrisierten Methode. Das Ergebnis ist also eine Klassen- bzw. Methodendefinition, die ebenso ein Programmierer

durch Kopieren und Ändern hätte erzeugen können.
Die Kopier-Sicht generischer Klassen und Methoden findet unter anderem in C++ in Form von sogenannten *Templates* Verwendung. Nach [Eck00] kopiert der C-Compiler im Prinzip nur den generischen Quellcode und ersetzt ähnlich wie bei der Verwendung von Makros einfach alle Typparameter durch konkrete Typen.
Die Klasse `DataPair` würde in C++ wie folgt implementiert werde.

```
template<class T>
class DataPair{
   private:
      T firstElem;
      T secondElem;
   public:
      DataPair(T first, T second);
      T getFirst();
      T getSecond();
      void swap();
};

template<class T>
DataPair<T>::DataPair(T first, T second){
   firstElem = first;
   secondElem = second;
}

template<class T>
T DataPair<T>::getFirst(){
   return firstElem;
}

template<class T>
T DataPair<T>::getSecond(){
   return secondElem;
}

template<class T>
void DataPair<T>::swap(){
   T help = firstElem;
   firstElem = secondElem;
   secondElem = help;
}
```

Beim Anlegen eines Paares von Personen-Objekten durch `DataPair<Person> pair = DataPair<Person>(Person(), Person())` würde der Code vom Compiler kopiert und folgendermaßen verändert werden.

```
class DataPair_Person{
   private:
      Person firstElem;
      Person secondElem;
   public:
      DataPair_Person(Person first, Person second);
```

```
      void setSecond(Person second);
      Person getSecond();
      void swap();
};

DataPair_Person::DataPair_Person(Person first, Person second){
   firstElem = first;
   secondElem = second;
}

Person DataPair_Person::getFirst(){
   return firstElem;
}

Person DataPair_Person::getSecond(){
   return secondElem;
}

void DataPair_Person::swap(){
   Person help = firstElem;
   firstElem = secondElem;
   secondElem = help;
}
```

Für ein Paar mit Autos würde er den Code erneut kopieren und wie folgt anpassen.

```
class DataPair_Car{
   private:
      Car firstElem;
      Car secondElem;
   public:
      DataPair_Car(Car first, Car second);
      void setSecond(Car second);
      Car getSecond();
      void swap();
};

DataPair_Car::DataPair_Car(Car first){
   firstElem = first;
   secondElem = second;
}

Car DataPair_Car::getFirst(){
   return firstElem;
}

Car DataPair_Car::getSecond(){
   return secondElem;
}

void DataPair_Car::swap(){
   Car help = firstElem;
   firstElem = secondElem;
```

```
   secondElem = help;
}
```

Diese Vorgehensweise mit Templates führt zu einer großen Anzahl neuer Klassen und Methodenimplementierungen. Es besteht also die Gefahr, dass sehr viel Binärcode entsteht, was nach [uKK04a] insbesondere dann reine Verschwendung ist, wenn die Collection-Klasse ausschließlich Pointer oder Referenzen auf Elemente verwaltet. Da ein Pointer bzw. eine Referenz stets denselben Speicherbedarf hat, unabhängig davon auf welche Art von Objekt sie verweist, ist es absolut unnötig für ein Paar von Personen-Objekten und ein Paar von Auto-Objekten seperaten Code zu erzeugen. Der generierte Binärcode für ein Paar von Personen-Referenzen ist nahezu identisch mit dem Binärcode für ein Paar von Auto-Referenzen. Der Unterschied liegt lediglich in einigen Typprüfungen und -konvertierungen, wenn Elemente zum Paar hinzugefügt oder aus dem Paar herausgeholt werden.

2.3.2 Generische Sicht und Java Generics

Bei dieser Variante wird für eine generische Klasse nur ein einziges Mal Zielcode erzeugt. Jede Instanzierung der parametrisierten Klasse arbeitet dann mit den gleichen Methodenimplementierungen, weshalb diese Variante nach [uKK04a] auch *Code-Sharing* bezeichnet wird im Gegensatz zu der als *Code-Specialization* bezeichneten Kopier-Sicht.
Code-Sharing ist nur möglich, wenn alle vordefinierten Datentypen und Objekte den gleichen Speicherbedarf haben, was durch die Einführung von einheitlichen Objektreferenzen in den übersetzten parametrisierten Klassen realisiert werden kann. Da in Java schon a priori alle Typen bis auf die Basistypen Referenztypen sind, ist es naheliegend, dass für die Übersetzung von parametrisierten Typen und Methoden in Java die Code-Sharing-Technik verwendet wird.

Die als *Java Generics* bezeichneten generischen Typen sind seit der Version 1.5 Bestandteil des JDK. Die Implementierung der generischen Liste sieht wie folgt aus.

```
class DataPair<T>{
   private T firstElem;
   private T secondElem;

   public DataPair(T first, T second){
      firstElem = first;
      secondElem = second;
   }

   public T getfirst(){
      return firstElem;
   }

   public T getSecond(){
      return secondElem;
   }

   public void swap(){
      T help = firstElem;
```

```
      firstElem = secondElem;
      secondElem = firstElem;
   }
}
```

Diese Typdeklaration wird nach [Bra04] nur ein einziges mal kompiliert und in ein einziges Class-File übersetzt, ganz genauso wie eine herkömmliche Java-Klasse. Es gibt keinerlei mehrfache Kopien des Codes. Weder im Quellcode, noch im Binärcode, noch auf der Platte oder im Speicher. Hierin besteht ein großer Unterschied zwischen C++ Templates und Java Generics. Darüber hinaus unterstützen die Java Generics Parameterrestriktionen, mit deren Hilfe definiert werden kann, dass der Typparameter einer Generischen Klasse ein bestimmtes Interface implementiert bzw. von einer bestimmten Super-Klasse abgeleitet ist. So wären folgende Klassendefinitionen denkbar.

```
class DataPair<T implements drivable>{
   ...
}

class DataPair<T extends Vehicle>{
   ...
}
```

Die Vorgehensweise des Java-Compilers basiert auf einer sogenannten *Type Erasure*, die - wie der Name schon sagt- jegliche generische Typinformation entfernt. Man kann sich wie in [uKK04a, uKK04b] beschrieben die Type Erasure als Übersetzung von generischem Java in reguläres Java vorstellen, bei der alle Typparameter durch die höchste Superklasse, die gerade noch den Paramterrestriktionen entspricht, ersetzt werden. In der Regel ist dies dann die Klasse `Object`. In Bezug auf das Paar würde dies bedeuten, dass die generische Klasse `DataPair<T>` in folgende herkömmliche Java-Klasse `DataPair` umgewandelt wird.

```
class DataPair{
   private Object firstElem;
   private Object secondElem;

   public DataPair(Object first, Object second){
      firstElem = first;
      secondElem = second:
   }

   public Object getFirst(){
      return firstElem;
   }

   public Object getSecond(){
      return secondElem;
   }

   public void swap(){
      Object help = firstElem;
      firstElem = secondElem;
      secondElem = firstElem;
```

```
   }
}
```

Dabei würden explizite Typkonvertierungen für den Programmierer wegfallen und vom Compiler selbst in den Binärcode eingefügt werden. So würde die Type Erasure aus

```
DataPair<Car> DataPair = new DataPair<Car>(new Car(), new Car());
DataPair.swap();
Car car = DataPair.getSecond();
```

automatisch den Code

```
DataPair DataPair = new DataPair(new Car(), new Car());
DataPair.swap();
Car car = (Car)DataPair.getSecond();
```

erzeugen. Zu beachten ist der benötigte Cast in der letzten Zeile, um die interne Referenz auf ein `Object` in eine Referenz auf ein `Car` umzuwandeln.

2.3.3 Typterme in Generic Java

Mit Einführung der Java Generics als Sprachelemente von Java Version 1.5 wurden einige Mängel der herkömmlichen Collection- und Container-Klassen in Java beseitigt. Einer der größten Mängel bestand in der Tatsache, dass diese Klassen nur Instanzen der Typs `Object` aufnehmen konnten. Für den Programmierer bot Java bis Version 1.4 keine Möglichkeit, den Typ der Container-Elemente näher zu spezifizieren. Dies hatte zur Folge, dass Java erhebliche Schwächen in der Typsicherheit aufwies. So aktzepierte der Java-Compiler beispielsweise folgende Anweisungen ohne Murren, die jedoch zur Laufzeit zu einer Typverletzung und somit zu einem Laufzeitfehler, genauer zu einer `ClassCastException` führten.

```
LinkedList myStringList = new LinkedList();
myStringList.add(new String());
Integer x = (Integer)myStringList.getFirst();
```

In Generic Java würde dieser Code-Schnippsel wie folgt aussehen.

```
LinkedList<String> myStringList = new LinkedList<String>();
myStringList.add(new String());
Integer x = myStringList.getFirst();
```

Hier kann der Programmierer also seine Absicht ausdrücken, eine Liste von Strings zu definieren, indem er über einen Typparameter die Art der Container-Element festlegt. Der Compiler erkennt schon beim Kompilieren den vermeintlichen Fehler in der letzten Zeile, der Programmierer kann den Fehler verbessern und die Typsicherheit zur Laufzeit ist gewährleistet.

Obwohl die explizite Angabe von Typparametern bei Collection- und Container-Klassen einen erheblichen Vorteil bietet, bringt diese auch einen Nachteil mit sich. So werden Typdeklarationen vor allem bei geschachtelten Containern sehr schnell unübersichtlich und für den Programmierer lästig.

Als Beispiel hierfür soll eine einfache Modellierung des Teils einer Bevölkerungsstatistik dienen. In einer verzeigerten Liste sollen die Ehepaare einer Kleinstadt über den Zeitraum von mehreren Jahren hinweg festgehalten werden. Die Deklaration einer solchen Liste sieht in regulärem Java folgendermaßen aus.

```
public LinkedList createCoupleStat(){
   LinkedList year_district_marriedCouple;
   ...
   return year_district_marriedCouple;
}
```

Da `LinkedList` ausschließlich Elemente vom Typ `Object` enthält, die nicht näher spezifiziert werden können, spielt es bei der Deklaration keine Rolle, welche Datentypen später in die Liste eingetragen werden. Somit kann ein Datenelement also durchaus selbst eine `LinkedList` sein, die wiederum Elemente des Typs `DataPair` enthält.
In Generic Java sieht die Sache etwas komplexer aus.

```
public LinkedList<LinkedList<Person, Person>> createCoupleStat(){
   LinkedList<LinkedList<Person, Person>> year_district_marriedCouple;
   ...
   return year_district_marriedCouple;
}
```

Hier müssen wie bei den Generics üblich, schon bei der Typdeklaration alle aktuellen Typparameter angegeben werden. Dieses einfache Beispiel soll zeigen, dass beim Anwenden von Java Generics recht schnell komplexe Typterme entstehen können, die den einen oder anderen Programmierer verwirren könnten. Das Beseitigen solcher Typterme ist Bestandteil der vorliegenden Studienarbeit. Durch Implementieren des Typinferenzalgorithmus soll es möglich sein, die zuvor erläuterte Deklaration wie folgt zu schreiben.

```
public createCoupleStat(){
   year_district_marriedCouple;
   ...
   return year_district_marriedCouple;
}
```

Der Typ der Variable `year_district_marriedCouple` und der Methode `createCoupleStat()` muss vom Programmierer nicht explizit angegeben werden, da der Compiler diesen mit Hilfe des Typinferenzsystems selbst berechnen kann. Dennoch wird die Typsicherheit zur Laufzeit sicher gestellt, da dem Compiler die Java Generics zugrunde liegen. Der Typinferenzalgorithmus vereint somit die Vorteile der alten und der neuen Java-Welt, indem er Programmierkomfort und Übersichtlichkeit auf der einen Seite und Typsicherheit auf der anderen Seite gewährleistet.

2.4 Typinferenz

Nach [Ehl04] stammt der Ausdruck Inferenz von dem englischen Wort „inference“ ab, welches man als Schlussfolgerung oder Rückschluss übersetzen kann. Typinferenz bedeutet demnach

Typrückschluss. In der Informatik versteht man unter Typinferenz das Folgern eines bestimmten Typs für einen gegebenen Ausdruck, obwohl der Typ nicht explizit angegeben wurde. Stattdessen wird unter Zuhilfenahme eines Algorithmus auf den richtigen Typ rückgeschlossen. Dieser Typinferenzalgorithmus arbeitet auf dem abstrakten Syntaxbaum und ist somit Teil der semantischen Analyse bzw. verkörpert mit einigen Erweiterungen versehen selbst die semantische Analyse.

Der in dieser Arbeit implementierte Typinferenzalgorithmus ist Bestandteil des Typinferenzsystems der von Professor Dr. Martin Plümicke im Zuge seiner Dissertation [Plü99] entwickelten Sprache *OBJ-P*. OBJ-P ist eine Erweiterung der funktionalen Programmiersprache *OBJ-3* um echten parametrisierten Polymorphismus, d.h. um echte generische Typen. In [Plü04a] wird das Typinferenzsystem von OBJ-P dann auf die Sprache Java übertragen.
Der Typinferenzalgorithmus `TRProg` rekonstrutiert Typen basierend auf dem in [Ott04] implementierten Unifikationalgorithmus `TUnify`. Die in `TUnify` durchgeführte Unifikation, wendet eine Typersetzung auf zwei Typterme an, um diese „gleichzumachen“. Diese mit Hilfe von *Schlussregeln* ermittelte Menge „gleichmachender“ Substitutionen, auch *Unifikator*, englisch „Unifier“, genannt, ist letztlich die Abbildung von Typparametern auf konkrete Typen.

Beispiel:

```
public int mul(int a, int b){
   return a * b;
}

public void foo(){
   x;
   x = mul(2,5);
}
```

Da bei der Deklaration der lokalen Variable `x` kein Typ angegeben wird, wird zunächst angenommen, dass `x` vom Typ `A` ist. Dabei ist `A` eine Typvariable.
Anhand der Methodensignatur der Methode `mul()` und der Zuweisung `x = mul(2,5)` kann jedoch geschlussfolgert werden, dass `x` vom Typ `Integer` sein muss. Diese Schlussfolgerung ist die Typinferenz. Die Menge der Typabbildungen $\{(A \rightarrow Integer)\}$ wird als Unifikator bezeichnet. Diese Menge besitzt durchaus nicht immer nur ein Element, wie das folgende Beispiel verdeutlichen soll.

```
public void bar(ht){
   Hashtable<String, Integer> table = new Hashtable<String, Integer>();
   if(ht.isEmpty()){
      ht = table;
   }
}
```

Zunächst ist der Typ des Methodenparameters `ht` unbekannt. Nach Untersuchung der Bedingung `ht.isEmpty()` wird auf Grund des Methodenzugehörigkeit von `isEmpty()` für `ht` der Typ `Hashtable<K, V>` angenommen. Dabei kann noch nichts über die Typparameter `K` und `V` ausgesagt werden.

Der Unifikationsalgorithmus liefert für die Zuweisung `ht = table` schließlich den Unifikator $\{(K \rightarrow String), (V \rightarrow Integer)\}$ zurück.

Kapitel 3

Anpassung des bestehenden Compilers

3.1 Analyse des Build Process

Für den Compilerbau bieten sich zahlreiche Toolkits und Code-Generatoren an, die es einem Entwickler erleichtern sollen, die für einen Compiler benötigten Bestandteile zu erstellen. So werden beim vorliegenden Java Compiler der Scannergenerator *JLex* sowie der Parsergenerator *jay* verwendet.
Dabei generiert JLex aus einem jlex-File, das die Lex-Spezifikation der Tokens in Form von regulären Ausdrücken enthält, den zugehörigen Scanner als Java-Programm. Analog erzeugt jay aus einem jay-File, das die erlaubten Tokens und die zugrunde liegende Grammatik spezifiziert, den jeweiligen Parser. Da diese beiden Tools die Java-Quellcodedateien für Scanner und Parser als Ausgabedaten haben, muss gewährleistet sein, dass diese Quellcodedateien stets bereits vor dem Ausführen des Java Compilers *javac* erzeugt werden. Dies wurde ursprünglich über ein *Makefile* gesteuert.

Um sowohl unter Linux als auch unter Windows entwickeln zu können und in Anbetracht der Tatsache, dass sowohl make als auch jay nur unter Linux lauffähig sind, ergaben sich daraus zwei Arten von Build Processes:

1. Unter Linux verwendete das Programm *make* zunächst JLex und jay, um danach dann den Sun Java Compiler javac zu starten.

2. Unter Windows kam das Programm *nmake* zum Einsatz, das zunächst JLex startete und dann anschließend unter einer Cygwin-Umgebung jay ausführte. Danach wurde dann die Windows-Version des Sun Java Compilers gestartet.

Um diese beiden Toolchains zu realisieren, wurden mehrere Makefiles, Shell-Scripts und Batch-Programme sowie eine Cygwin-Umgebung unter Windows eingesetzt.

3.2 Neustrukturierung des Softwareprojektes

Der zu Projektbeginn übernommene Java Compiler bestand aus 132 Quellcode-Dateien, die alle im selben Verzeichnis abgelegt waren. Auf Grund dieser Fülle an ungeordneten Software-Modulen ließ sich die bestehende Software recht schlecht überblicken und analysieren. Darüber hinaus war es wünschenswert das Projekt in eine professionelle Entwicklungsumgebung zu überführen, die das Programmieren und Debuggen der Software erleichtert.

Aus diesen Gründen wurde in der Java-Entwicklungsumgebung *Eclipse* ein komplett neues Java-Projekt namens „JavaCompilerCore“ erstellt, in das die bestehenden Quellcode-Dateien überführt wurden. Dabei wurden die Software-Module in folgende Packages gegliedert:

```
mycompiler
mycompiler.mybytecode
mycompiler.myclass
mycompiler.myexception
mycompiler.mymodifier
mycompiler.myoperator
mycompiler.myparser
mycompiler.mystatement
mycompiler.mytest
mycompiler.mytype
mycompiler.mytypereconstruction
mycompiler.mytypereconstruction.replacementlistener
mycompiler.mytypereconstruction.set
mycompiler.mytypereconstruction.typeassumption
mycompiler.mytypereconstruction.typeassumptionkey
mycompiler.mytypereconstruction.unify
mycompiler.unused
```

Der Build-Process wurde durch die beiden Ant-Scripts „AntParserBuilderLinux.xml“ und „AntParserBuilder.xml“ gekapselt und ist nun in die Entwicklunsgumgebung integriert. Da Eclipse eine Java-Anwendung ist, kann sowohl unter Linux als auch unter Windows entwickelt werden. Beim Wechsel des Betriebssystems muss lediglich in den Projekteinstellungen das Ant-Script ausgetauscht werden.

Darüber hinaus wurden alle für das Projekt relevanten Dokumente und Tools in das Projekt aufgenommen. Folgende Unterverzeichnisse umfasst das Projekt:

- **bin**: Die kompilierten Binärdateien
- **doc**: Dokumentation wie UML-Diagramme, JavaDoc und Vorgänger-Studienarbeiten
- **examples**: Beispielprogramme und Use Cases
- **notizen**: Notizen, To-Do-Listen, etc.

- **src**: Der Quellcode
- **tools**: Alle für den Build Process benötigten Tools wie JLex, jay, Cygwin, usw.

Weitere Informationen zum Projektaufbau und zur Einrichtung und Konfiguration der Entwicklungsumgebung Eclipse finden sich im → Anhang A.

3.3 Änderungen am Quellcode

Nach ausführlicher Analyse der vorliegenden UML-Diagramme und des Quellcodes wurde deutlich, dass vor der Implementierung des Typinferenzalgorithmus zunächst einige Änderungen am Quellcode vorgenommen werden mussten.
Da die Weiterentwicklung eines Softwareprojekts der vorliegenden Größenordnung es zwangsläufig mit sich bringt, kontinuierlich älteren Quellcode anzupassen und zu erweitern, würde es den Rahmen dieser Dokumentation sprengen, jede einzelne Änderung aufzuführen. Vielmehr sollen an dieser Stelle einige wenige, weitreichende Änderungen kurz umrissen werden, die für das Design der Software entscheidend sind.

3.3.1 TyploseVariable vs. GenericTypeVar

Bei der Implementierung des Typinferenzsystems müssen zwei Arten von Typparametern unterschieden werden.

Zunächst einmal gibt es die Typparameter der Java Generics, die bei Klassendefinitionen explizit angegeben werden müssen. Sie dienen wie bereits in den theoretischen Grundlagen beschrieben dazu, generische Datentypen zu erzeugen, deren Operationen nicht für jede Typinstanz neu programmiert werden müssen. Diese Typparameter sind fester Bestandteil der Sprache Java seit JDK 1.5 und werden im folgenden als *generische Typvariablen* bezeichnet werden.
Davon zu unterscheiden sind diejenigen Typparameter, die als Platzhalter für die vom Programmierer nicht angegebenen Typen dienen. Diese Art von Typparameter hat zunächst einmal nichts mit der Sprache Java oder den Java Generics zu tun, sondern ist lediglich ein Hilfsdatentyp im zu implementierenden Typinferenzsystem. Ein solcher Typparameter wird von nun an als *typlose Variable* bezeichnet.

Bezüglich der Typparameter war die Implementierung des bestehenden Java Compilers nicht ganz korrekt. So wurde kein Unterschied zwischen den beiden Arten von Typparametern gemacht. Viel mehr wurden vom bestehenden Parser sowohl für GenericTypeVars als auch für TyploseVariablen Objekte der Klasse `TyploseVariable` erzeugt. Dies würde bei der Implementierung des TRA letztlich zu Problemen führen, da diese beiden Arten von Typparametern an einigen Stellen syntaktisch wie semantisch unterschiedlich behandelt werden müssen.

Um diesen Fehler zu beheben wurde die Klasse `GenericTypeVar`, abgeleitet von `Type`, eingeführt. An den entsprechenden Stellen im jay-File und im Quellcode werden nun anstatt

Objekte des Typs TyploseVariable Objekte des Typs GenericTypeVar erzeugt bzw. erwartet. Darüber hinaus war die im jay-File spezifizierte Grammatik unvollständig, da sie es nur erlaubt hat, bei Methodenparametern die Typangabe wegzulassen. Die fehlenden Grammatikregeln wurden hinzugefügt, sodass nun an allen relevanten Stellen typlose Variablen erzeugt werden. Ergänzend dazu wurde die Klasse `TyploseVariable` überarbeitet und erweitert, um den Anforderungen des Typinferenzalgorithmus gerecht zu werden → Kapitel 4.3.

3.3.2 Einführung von BaseTypes

Obwohl bereits Klassen für die BaseTypes angelegt waren, wurden diese nirgendwo erzeugt, sondern stattdessen die korrespondierenden RefTypes verwendet. Dies wurde durch Korrekturen im jay-File und im Quellcode geändert, sodass der abstrakte Syntaxbaum nun BaseTypes enthält.

Da jedoch der Algorithmus `TUnify` und die für die Bytecodegenerierung zuständigen Methoden noch immer keine BaseTypes unterstützen, mussten bei der Implementierung des TRA einige Kompromisse eingegangen werden. So werden an bestimmten Stellen in den Algorithmen `TRExp`, die speziell gekennzeichnet und kommentiert sind, als Workaround noch immer RefTypes anstelle von BaseTypes verwendet.

3.3.3 Korrekturen bei der Verwendung des Typs Void

Ähnlich wie bei den BaseTypes war im bestehenden Compiler bereits die Klasse `Void` angelegt, wurde jedoch beim Parsen niemals instanziert. Stattdessen wurden RefTypes mit Namen „void“ erzeugt. Durch entsprechende Änderungen am jay-File wurde auch dies behoben.

3.3.4 Einführung von Zeilennummern

Im Hinblick auf Partnerstudienarbeit und deren Integration des Compilers in die IDE ist es äußerst wichtig, dass über die zu entwickelnde Schnittstelle alle relevanten Informationen über den abstrakten Syntaxbaum abrufbar sind. Sollen bei der Entwicklung eines Editor-Plugins Features wie Status-Anzeige, Markierungen oder Kontextmenüs zu den einzelnen Deklarationen implementiert werden, so müssen diese Objekte des Syntaxbaumes die Zeilennummer als Feldvariable enthalten. Nur so kann eine Kopplung von abstraktem Syntaxbaum zu Editorquellcode realisiert werden.

Aus diesem Grund wurde die Klasse `Token` um das Member `m_LineNumber` erweitert und das lex-File sowie das jay-File der Art modifiziert, dass beim Parsen die Zeilennummer an die erzeugten Objekte weitergegeben wird.

Kapitel 4

Der Typinferenzalgorithmus

Der implementierte Algorithmus besteht insgesamt aus ca. 40 Unteralgorithmen, deren theoretische Beschreibung aus [Plü04a] in den → Anhang C übernommen worden ist.

Im Folgenden soll nun ein kleiner Überblick über die wichtigsten Datenstrukturen und Methoden gegeben werden, sodass die groben Zusammenhänge und Strukturen deutlich werden. Dieses Kapitel ist als begleitende Dokumentation zu den Kommentaren im Quellcode und den Ausführungen in [Plü04a] zu sehen. Für tiefere Einblicke in die Implementierung liegt dieser Arbeit eine CD-Rom bei, auf der das komplette Projekt inklusive Javadoc und UML-Diagramme zu finden ist.

Zur weiteren Anschauung soll folgendes Beispielprogramm dienen:

```
public class UselessFoo<GTV_1, GTV_2> {
   public mul(Matrix m, int n){
      m.scalarMul(n);
      return m;
   }

   public void doSomething(container){
      Matrix matrix = new Matrix();
      Matrix result;
      number = 5+2;

      result = mul(matrix, number);
      container.addElement(result);
   }
}
```

In diesem Java-Programm sind die Typen der Methode `mul()`, des Methodenparameters `container` sowie der lokalen Variablen `number` nicht angegeben und somit zunächst unbekannt. Demzufolge werden für diese Identifier typlose Variablen angenommen.

4.1 Allgemeine Beschreibung

Der TRA hat die Aufgabe, den Quellcode bzw. den abstrakten Syntaxbaum rekursiv auf Typinformationen hin zu analysieren und diese so miteinander zu verknüpfen, dass am Ende feststeht, welche Variable bzw. Methode welchen Typ besitzt. Dabei werden während des gesamten Ablaufs Typannahmen getroffen, die nach und nach verbessert oder wieder verworfen werden.

Viele Statements im Quellcode sind was den Typ angeht nicht eindeutig und lassen mehrere Typrückschlüsse zu. Der Methodenaufruf `container.addElement(result)` des UselessFoo-Beispiels soll dies kurz verdeutlichen. Der Aufruf lässt auf Grund der Typhierarchie die Annahmen zu, dass `container` den Typ `Vector` oder den Typ `Stack`, der von `Vector` abgeleitet ist, hat. Der Typ lässt sich nicht präzisieren, da beide Klassen die Methode `addElement()` besitzen.

Jede dieser möglichen Typannahmen muss der TRA mit bereits bestehenden Typannahmen kombinieren und als getrennte Vorraussetzungen für neue Annahmen in seine Wissensbasis aufnehmen. Prinzipiell dienen dem Algorithmus in jedem Zyklus die bestehenden Typannahmen als Eingabedaten, auf deren Basis die neuen Typinformationen des aktuellen Statements berechnet werden. Je weiter der Algorithmus fortschreitet, desto mehr mögliche Typkombinationen entstehen bzw. werden wieder verworfen. So präzisiert der Algorithmus nach und nach seine Typinformationen.
Am Ende liegt als Ergebnis des TRA eine Menge von möglichen Typkombinationen vor, von denen der Anwender die gewünschte auswählen kann.

4.2 Grundlegende Datenstrukturen

4.2.1 Die Klasse CTypeAssumption

Diese Klasse ist die Hauptdatenstruktur auf der der TRA arbeitet. Sie repräsentiert eine Abbildung eines Bezeichners auf einen angenommenen Datentyp. `CTypeAssumption` selbst ist eine abstrakte Basisklasse, die für eine solche allgemeine Typannahme steht. Die von ihr abgeleiteten Subklassen stehen dann für bestimmte Typannahmen im abstrakten Syntaxbaum:

Klasse	Bedeutung
`CInstVarTypeAssumption`	Typanahme für eine Feldvariable
`CMethodTypeAssumption`	Typanahme für eine Methode; enthält zusätzliche Typannahmen für Methodenparameter
`CParaTypeAssumption`	Typanahme für einen Methodenparameter
`CLocalVarTypeAssumption`	Typanahme für eine lokale Variable

Zu Beginn des TRA werden `CTypeAssumptions` für alle Felddeklarationen (Feldvariablen und Methoden) einer Klasse erzeugt und zu einer Menge von Typannahmen (→ Kapitel 4.2.5) zusammengefasst. Während der Abarbeitung des Algorithmus entstehen dann immer neue

Typerkenntnisse, die in Form von neuen `CTypeAssumptions` zu dieser Menge hinzugefügt werden oder aber durch Modifikation der bestehenden `CTypeAssumptions` in die Menge eingehen.

So würde im Beispielprogramm diese Menge zu Beginn des TRA die `CMethodTypeAssumption` $mul : Matrix \times Integer \rightarrow A$ enthalten, nach der Unifizierung der Rückgabetypen jedoch die modifizierte Typannahme $mul : Matrix \times Integer \rightarrow Matrix$.

4.2.2 Die Klasse CTypeAssumptionKey

Um das Zwischenspeichern und Suchen nach Typannahmen zu erleichtern, wird in der vorliegenden Implementierung reger Gebrauch von Hashtables gemacht. Diese legen Referenzen auf Typannahmen unter einem eindeutigen Schlüssel ab, über den die Referenz später dann wieder herausgesucht werden kann. Aus diesem Grund musste eine spezielle Notation eingeführt werden, die als eindeutige Schlüsselzeichenkette für eine bestimmte Typannahme dient.

Typannahme	Notation
InstVarTypeA.	**InstVar#***Klassenname***.***Identifier*
MethodTypeA.	**Method#***Klassenname***.***Identifier***(***AnzahlParameter***)**
ParaTypeA.	**LocalVar#***Klassenname***.***Methodenname***(***AnzahlParameter***).***Identifier*
LocalVarTypeA.	**LocalVar#***Klassenname***.***Methodenname***(***AnzahlParameter***).Block_***BlockID***.***Identifier*

Hierbei sind die kursiv gedruckten Teile variabel zu sehen, die je nach konkreter Typannahme variieren.
So würde beispielsweise für die lokale Variable `number` aus dem Math-Beispiel eine `CLocalVarTypeAssumption` mit dem Schlüsselwert `LocalVar#Math.operation(0).Block_1.number` erzeugt werden.

Um diese Notation zu kapseln und somit ständige String-Operationen zu vermeiden, wurde eine Art Namensdienst implementiert. Dieser beruht hauptsächlich auf einer parallel zur `CTypeAssumption`-Typhierarchie aufgebauten Datenstruktur, die die Schlüsselwerte modelliert. Abgeleitet von der abstrakten Superklasse `CTypeAssumptionKey` wurden folgende Klassen implementiert:

- `CInstVarKey`
- `CMethodKey`
- `CMethodParaKey`
- `CLocalVarKey`

Diese einzelnen Klassen erhalten über Ihren Konstruktor alle relevanten Bezeichner, um die jeweilige Typannahme eindeutig zu spezifizieren, und erzeugen intern dann die Schlüsselzeichenkette, die als Feldvariable gespeichert wird. Die von `Object` vererbte Methode `int hashcode()` wird von `CTypeAssumptionKey` so überschrieben, dass sie den eindeutigen Hashwert der Schlüsselzeichenkette zurückliefert.

Somit muss also nur ein `CTypeAssumptionKey` erzeugt und dieser an die Hashtable übergeben werden, um die zugehörige `CTypeAssumption` zu erhalten. Dies erleichtert den Umgang mit den Typannahmen erheblich.

4.2.3 Die Klasse CSubstitution

Diese Klasse modelliert die Ersetzung einer Instanz von `TyploseVariable` und ist somit die Abbildung einer typlosen Variable auf einen ermittelnden Typ, durch den diese ersetzt werden soll. Darüberhinaus stellt diese Klasse über Ihre Methode `execute()` die eigentliche Typersetzung im abstrakten Syntaxbaum als Operation zur Verfügung (→ Kapitel 4.3) und ist somit neben `CTypeAssumption` die für diese Implementierung des TRA wichtigste Datenstruktur.

In Bezug auf unser UselessFoo-Beispiel, könnte am Ende des TRA folgende Menge von `CSubstitution` vorliegen: $\{(A \rightarrow Matrix), (B \rightarrow Vector), (C \rightarrow Integer)\}$.

Instanzen von `CSubstitution` werden meist mit Daten aus `Pair`-Objekten initialisiert.

4.2.4 Die Klasse Pair

Die Klasse `Pair` ist die im Rahmen von [Ott04] entwickelte Implementierung einer Typabbildung der Art $TypA \rightarrow TypB$, die vom Unifikationsalgorithmus `TUnify` intern benutzt wird. Eine Menge von `Pair`-Objekten modelliert somit den unter → Kapitel 2.4 beschriebenen Unifikator.

Die Klasse `Pair` ist der Klasse `CSubstitution` prinzipiell sehr ähnlich, wird jedoch in einem anderen Kontext verwendet und stellt andere Operationen zur Verfügung. In der Regel wird über die Ausgabedaten von `TUnify` iteriert und für die weitere Verarbeitung aus jedem `Pair` eine `CSubstitution` erzeugt.

4.2.5 Implementierung von Mengen

Da die Standard-Collection-Klassen aus dem Package `java.util` in vielen Punkten nicht ausreichend erschienen, um im Sinne der Mengenalgebra eine Menge von Objekten zu modellieren, wurde eine eigene Implementierung vorgezogen.

Dabei definiert die abstrakte Superklasse `CSet<E>` die Schnittstelle zu jeglicher Art von Datenmenge in Form von Mengenoperationen wie `unite()`, `substract()`, `shallowCopy()` oder `deepCopy()`. Durch diese Schnittstelle wird der Zugriff auf die Menge von der tatsächlichen Implementierung der Containerklasse entkoppelt. Darüberhinaus ist `CSet` eine generische Klasse, die Datenobjekte des Typparameters `E` enthält. Somit können also Mengen eines beliebigen Datentyps erzeugt werden.

Konkrete Subklassen von `CSet` sind die Klassen `CVectorSet<E>` und `CHashtableSet<E>`, die aufbauend auf einem `Vector` bzw. einer `Hashtable` die abstrakten Mengenoperationen

aus `CSet` implementieren.
Um bei `CHashtableSet` den Umgang mit Datenelementen und deren Schlüsseln zu erleichtern, wurde diese Klasse mit Parameterrestriktionen versehen. So müssen Elemente der Menge das Interface `IHashSetElement` und deren Schlüssel das Interface `IHashSetKey` implementieren. Über diese Schnittstellen kann dann beispielsweise bei Mengen von Typannahmen die automatische Schlüsselgenerierung (→ Kapitel 4.2.2) ausgeführt werden.

4.2.6 Die Klasse CIntersectionType

Die Klasse `CIntersectionType` ist lediglich eine Wrapper-Class für eine `CMethodTypeAssumption`, die auf Wunsch von Professor Dr. Plümicke eingeführt wurde, um zukünftige Erweiterungen der im nächsten Abschnitt beschriebenen Ergebnisdatenstruktur zu erleichtern. `CIntersectionType` besteht letztlich nur aus einem `Vector` von `CMethodTypeAssumption`, der ein einziges Element enthält, und dem zu diesem Element gehörenden `CMethodKey`.

4.2.7 Die Ergebnisdatenstruktur CTypeReconstructionResult

Wie bereits in → Kapitel 4.1 erläutert, liefert der TRA eine Menge von möglichen Typkombinationen für das vorliegende Java-Programm zurück. Diese Menge ist in Form eines `Vector` von `CTypeReconstructionResult` implementiert. `CTypeReconstructionResult` kapselt eine mögliche Typkombination in Form von `CTypeAssumption`-, `CIntersectionType`- und `CSubstitution`-Instanzen.
Die Menge aller Methodentypannahmen ist als `Hashtable` `m_MethodIntersectionTypes` gespeichert, die Typannahmen für lokale Variablen und Feldvariablen in der `Hashtable` `m_FieldAndLocalVarAssumptions`. Die zu diesen Typannahmen gehörenden Substitutionen sind im Feld `m_Substitutions` abgelegt. Als Zusatzinformation werden noch alle generischen Typvariablen nach ihren Klassen sortiert in der `Hashtable` `m_GenericTypeVars` sowie alle im abstrakten Syntaxbaum vorkommenden Klassen im `Vector` `m_ClassNames` mitgeliefert.

Für das UselessFoo-Beispiel würde die Ergebnisdatenstruktur aus einem `Vector` mit zwei `CTypeReconstructionResult` bestehen.

1. `m_MethodIntersectionTypes`:
   $\{(mul : Matrix \times Integer \rightarrow Matrix), (doSomething : Vector \rightarrow Void)\}$

   `m_FieldAndLocalVarAssumptions`:
   $\{(matrix : Matrix), (result : Matrix), (number : Integer)\}$

   `m_Substitutions`:
   $\{(A \rightarrow Matrix), (B \rightarrow Vector), (C \rightarrow Integer)\}$

   `m_GenericTypeVars`:
   $\{GTV\_1, GTV\_2\}$

2. `m_MethodIntersectionTypes`:
   $\{(mul : Matrix \times Integer \rightarrow Matrix), (doSomething : Stack \rightarrow Void)\}$

   `m_FieldAndLocalVarAssumptions`:
   $\{(matrix : Matrix), (result : Matrix), (number : Integer)\}$

   `m_Substitutions`:
   $\{(A \rightarrow Matrix), (B \rightarrow Stack), (C \rightarrow Integer)\}$

   `m_GenericTypeVars`:
   $\{GTV\_1, GTV\_2\}$

4.3 Kopplung von Algorithmus und Syntaxbaum

Der TRA arbeitet nach [Plü04a] ausschließlich syntaxbasiert, d.h. er greift ausschließlich lesend auf den Syntaxbaum zu; die konkreten Objekte im Speicher sind unbedeutend für ihn. Es kommt nur auf die Inhalte dieser Objekte, d.h. die geparste Syntax an, und seine Ergebnisdatenstruktur besitzt keinerlei eindeutig zuordenbare Referenzen auf Elemente des Syntaxbaumes. Diese Tatsache erleichtert zwar die Implementierung des TRA, führt jedoch nach Abschluss des Algorithmus zu einem recht großen Problem: Wie sollen die berechneten Substitutionen der Ergebnisdaten auf die konkreten typlosen Variablen des Syntaxbaumes angewendet werden, wenn sie völlig losgelöst davon sind?

Der erste Lösungweg für dieses Problems, der sich anbot, war das Schreiben eines zweiten Algorithmus, der nach Abschluss des TRA noch einmal in den abstrakten Syntaxbaum absteigen und alle `TyploseVariable`-Objekte gemäß der berechneten Substitutionen gegen andere Typ-Objekte austauschen würde. Diese Lösung wurde jedoch recht schnell verworfen, da sie mit einem erheblichen Aufwand an Programmierarbeit verbunden gewesen wäre.

Ein weitere Lösungsgedanke bestand darin, die Substitutionen schon während des TRA auszuführen. Das Problem hierbei bestand nur darin, dass die Änderungen am abstrakten Syntaxbaum die weiteren Typannahmen des TRA verfälscht hätten. Darüber hinaus hätten auf Grund der unter → Kapitel 4.1 beschriebenen Nichteindeutigkeit mehrere Syntaxbäume parallel gehalten werden müssen. Dies hätte sich nur durch Klonen des gesamten Syntaxbaumes realisieren lassen können, was sowohl aus Gründen der Performanz als auch aus Gründen des Programmieraufwandes äußerst ungünstig gewesen wäre.

Letztlich wurde getreu dem Motto KISS - „Keep it simple stupid!“ - eine Lösung gewählt, die einfach zu implementieren war und sich als äußerst performant erwies. Diese Lösung beruht auf dem in [oF95] beschriebenen Observer-Pattern, nach dem sich viele *Observer*, auch *Listener* genannt, bei einem Objekt, für das sie sich interessieren, registrieren, um zukünftig über dessen Statusänderungen benachrichtigt zu werden.

Im vorliegenden Fall registriert sich ein Element des abstrakten Syntaxbaumes bei der Zuweisung eines `TyploseVariable`-Objektes bei diesem als Observer, der sich für diese typlose Variable interessiert. Ein solcher Observer bzw. Listener wird durch das Interface `ITypeReplacementListener` beschrieben und kann über dessen Interface-Methode `replaceType( CReplaceTypeEvent e)` über eine „Statusänderung", d.h. eine Substitution, informiert werden. So kann er seine `TyploseVariable` durch den Typ ersetzt, den er über das `CReplaceTypeEvent` mitgeteilt bekommt. Folgende Klassen implementieren das Interface `ITypeReplacementListener` und können sich als Listener eintragen:

- InstVarDecl
- LocalVarDecl
- FormalParameter
- Method
- ExprStmt

Die `TyploseVariable` registriert Ihre `ITypeReplacementListener` im `Vector m_ReplacementListeners` und feuert bei Aufruf der Methode `fireReplaceTypeEvent()` ein Ereignis ab, das an alle eingetragenen Listener geht.

Obwohl diese Lösung auf den ersten Blick äußerst einfach zu realisieren schien, zeichnete sich recht schnell ein Folgeproblem ab. Damit die Observer-Technik korrekt funktioniert, müssen sich alle Observer, die sich für die typlose Variable A interessieren, bei ein und derselben `TyploseVariable`-Instanz für A eintragen. Das bedeutet, dass es innerhalb des abstrakten Syntaxbaumes niemals zwei Instanzen für die typlose Variable A geben darf. Um dies zu gewährleisten, wurde das direkte Erzeugen von `TyploseVariable`-Objekten unterbunden. Stattdessen existiert nun die statische Factory-Methode [oF95] `TyploseVariable.fresh()`, die eine neue `TyploseVariable` erzeugt und in einer zentralen Registry einträgt. Über eine weitere statische Methode `getInstance(String name)` kann eine bestehende `TyploseVariable` aus der Registry ausgelesen werden.

Damit gestaltet sich die Umsetzung des TRA wie folgt. Der TRA berechnet alle möglichen Typkombinationen und liefert die zugehörigen Mengen von Substitutionen in `CTypeReconstructionResults` verpackt zurück. Über die IDE wird vom Anwender die gewünschte Typkombination ausgewählt und die zugehörige Menge an `CSubstitution` selektiert. Über diese Menge wird daraufhin iteriert und für jede einzelne `CSubstitution` die Methode `execute()` aufgerufen, die dann die eindeutige `TyploseVariable`-Instanz aus der Registry heraussucht und darauf `fireReplaceTypeEvent()` aufruft. Auf diese Art werden im gesamten Syntaxbaum alle typlosen Variablen durch die berechneten Typen ersetzt.

4.4 Substitutionsbasiertes Konzept

Der in [Plü04a] beschriebene TRA arbeitet mit einem auf Typannahmen beruhenden Prinzip. Der syntaxbasierte TRA sammelt Typinformationen und Unifikationsergebnisse, um damit bestehene Typannahmen zu präzisieren und zu erweitern. Typsubstitutionen spielen dabei eine untergeordnete Rolle und sind stets nur Mittel zum Zweck. Die Substitutionen der Unifikatoren, die der Unifikationsalgorithmus zurückliefert, werden auf die Mengen von Typannahmen ausgeführt und danach meist wieder verworfen.
Am Ende zählt nur die Menge der Typannahmen, die ein theoretisches Typabbild für das gesamte Java-Programm verkörpert.

Dieses Prinzip ist für eine Implementierung gänzlich ungeeignet, da ein solches Gesamtabbild als Ergebnis nicht auf den abstrakten Syntaxbaum im Speicher angewendet werden kann. Viel mehr müssen diese Typinformationen wieder in kleine, ausführbare Anweisungen heruntergebrochen werden, die auf den Syntaxbaum anwendbar sind. Diese kleinen Einzelanweisungen sind aber genau die Typsubstitutionen. Abweichend von den Ausführungen in [Plü04a] macht es also durchaus Sinn, sich grundsätzlich alle Substitutionen als Mengen von `CSubstituion` zu merken und innerhalb des Algorithmus weiterzugeben.
Für die vorliegende Implementierung ist aussschließlich die Menge der Substitutionen in der Ergebnisdatenstruktur des TRA (→ Kapitel 4.2.7) von Bedeutung. Die Menge der Typannahmen dient lediglich zu Anschauungszwecken, um dem Anwender eine Gesamtabbild der berechneten Typinformationen zur Verfügung zu stellen.

Um dieses von [Plü04a] substitutionsbasierte Konzept in Verbindung mit der Problematik aus → Kapitel 4.3 implementieren zu können, müssen auf drei unterschiedlichen Ebenen Substitutionen vorgenommen werden:

1. Während des TRA auf den Mengen von Typannahmen über die Methode `CTypeAssumption.sub()`

   Beispiel.:
   Bestehende Menge von Typannahmen: $V = \{number : C\}$
   Neu berechneter Unifier: $\sigma_{new} = \{(A \rightarrow Matrix), (C \rightarrow Integer)\}$
   Anwendung von σ auf V ergibt: $V = \{number : Integer\}$

2. Während des TRA auf den Mengen von bereits ermittelten Substitutionen über die Methode `CSubstitution.applyUnifier()`

   Beispiel.:
   Bestehende Menge von Substitutionen: $\sigma = \{D \rightarrow C\}$
   Neu berechneter Unifier: $\sigma_{new} = \{(A \rightarrow Matrix), (C \rightarrow Integer)\}$
   Anwendung von σ_{new} auf σ ergibt: $\sigma = \{(D \rightarrow Integer)\}$

3. Nach Abschluss des TRA auf dem abstrakten Syntaxbaum über die Methode `CSubstitution.execute()` ⇒ Permanente Änderung im Speicher

4.5 Die Klasse GenericTypeVar

Wie bereits in → Kapitel 3.3.1 erwähnt, müssen an einigen Stellen des Compilers respektive des TRA die generischen Typvariablen anders behandelt werden als die typlosen Variablen.
So gelten während des TRA innerhalb einer generischen Klasse alle ihre generischen Typvariablen als konstante Typparameter, die nicht ersetzt werden dürfen. Außerhalb der Klasse werden dieselben Typvariablen jedoch als typlose Variablen behandelt, die bei der Instanzierung der generischen Klasse durch konkrete Typen ersetzt werden.

Dies soll am Beispiel der Klasse `MyVector<E>` kurz verdeutlicht werden.

```
class MyVector<E>{
   void addElement(obj){
   ...
   }
}

class MyApplication{
   void main(String[] args){
      MyVector<Integer> v = new Vector<Integer>();
      v.addElement(new Integer(4));
   }
}
```

Hier darf der TRA, solange er die Typen der Klasse `MyVector<E>` rekonstruiert, die Typvariable `E` nicht durch andere Typen ersetzen, da sie als fester, unveränderlicher Parameter für eine spätere Typinstanz steht. `E` wird als `GenericTypeVar` betrachtet.
Ist die Typrekonstruktion der Klasse `MyVector<E>` abgeschlossen und arbeitet der TRA dann auf `MyApplication`, muss die Typvariable `E` veränderbar sein. Für `E` kann nun jeder beliebige Typ instanziert, werden. `E` wird hier als `TyploseVariable` verwendet.

Ein weiterer Punkt ist das Schliessen von `TyploseVariable`-Instanzen, über die am Ende des TRA keine Typinformation vorliegt, auf vorhandene `GenericTypeVars`. So sollte am Ende des TRAs und nach Ausführen aller Substitutionen keine `TyploseVariable` mehr im abstrakten Syntaxbaum vorhanden sein. Auf das Beispiel `MyVector` übertragen hieße dies, dass die `TyploseVariable` für den Parameter `obj` der Methode `addElement()` nach Ende des TRAs für `MyVector` durch die `GenericTypeVar` `E` ersetzt werden müsste.

Die letztendliche Umsetzung dieser Unterscheidungen zwischen generischen Typparametern und typlosen Variablen wurde noch nicht vollständig realisiert und sollte in einer der Nachfolgerstudienarbeiten implementiert werden. Momentan erzeugt der Parser nur für die Typterme im Kopf der Klassendefinition `GenericTypeVars`. Ansonsten werden ausschließlich `TyploseVariable`-Objekte verwendet.
Die oben beschriebene Zuordnung am Ende des TRAs erfolgt derzeit per manueller Auswahl des Anwenders über die IDE-Schnittstelle. Dieser wählt aus, welche `TyploseVariable` durch welche `GenericTypeVar` ersetzt werden soll.

4.6 Das IDE-Interface

Die Schnittstelle zum entwickelten Compiler wird über das Java-Interface `MyCompilerAPI` zur Verfügung gestellt. Diese Schnittstelle stellt für das IDE-Plugin der Partnerstudienarbeit alle relevanten Methoden zu Verfügung, um die Funktionalität des Compilers zu nutzen und auf dessen Ergebnisdaten zuzugreifen.
Über die statische Factory-Methode [oF95] `MyCompiler.getAPI()` wird eine Singleton-Instanz [oF95] des Compilers erzeugt und als `MyCompilerAPI`-Instanz zurückgeliefert. Anschließend besteht Zugriff auf folgende Interface-Methoden:

- `void init()`: Initialisiert den Compiler. Sollte vor jedem neuen Parsvorgang aufgerufen werden.
- `boolean setDebugLevel(int debugLevel)`: Legt den Debug-Level fest, der die Debugging-Ausgabe für Module der Vorgängerstudienarbeiten steuert.
- `void parse(File file)`: Parst eine Quellcodedatei und baut einen abstrakten Syntaxbaum auf.
- `void parse(String srcCode)`: Parst eine Quellcode-Zeichenkette und baut einen abstrakten Syntaxbaum auf.
- `void semanticCheck()`: Ruft die semantische Analyse der Vorgängerstudienarbeit ohne Typrekonstruktion auf. Diese Methode wurde nur der Vollständigkeit wegen aufgenommen und sollte nicht mehr verwendet werden.
- `Vector<CTypeReconstructionResult> typeReconstruction()`: Ruft den Typrekonstruktionsalgorithmus auf, der die alte semantische Analyse ersetzt.
- `SourceFile getSyntaxTree()`: Liefert den geparsten Syntaxbaum zurück.
- `void codeGeneration()`: Generiert den Bytecode und das Class-File für den Syntaxbaum.

Für die Verarbeitung der Ergebnisdatenmenge des TRA, die von `typeReonstruction()` zurückgeliefert wird, werden keine speziellen Interface-Methoden bereitgestellt. Es bleibt dem IDE-Programmier selbst überlassen, wie und wann er die Daten durchsucht, darstellt und mittels `CSubstitution.execute()` umsetzt.
Wie bereits in → Kapitel 4.4 kurz angedeutet, war der in [Plü04a] beschriebene Algorithmus nie darauf ausgelegt, eine möglichst einfache Umsetzung der Ergebnisdaten zu gewährleisten, geschweige denn eine einfache Benutzerschnittstelle zu diesen Daten zur Verfügung zu stellen. Vielmehr bestand die Intension darin, die Typinformation für ein Java-Programm in ihrer Gesamtheit zu berechnen. Dieser Umstand mag einen IDE-Programmierer, der eine im Sinne der Softwareergonomie benutzerfreundliche Schnittstelle schaffen möchte, wohl vor eine recht knifflige Aufgabe stellen.

4.7 Exception Handling

Für die Implementierung des TRA wurde die von `RuntimeException` abgeleitete Fehlerklasse `CTypeReconstructionException` eingeführt. Bei jeglichem Typ- bzw. Semantikfehler, der im Rahmen des TRA auftritt wird solch eine Exception erzeugt und bis zur Interface-Methode `typeReconstruction()` durchgereicht, sodass die IDE den Fehler anzeigen kann.

4.8 Unteralgorithmen

4.8.1 Die Einstiegsmethode typeReconstruction()

Die Einstiegsmethode in den TRA, die von der Interface-Methode der `MyCompilerAPI` gerufen wird, ist die Methode `SourceFile.typeReconstruction()`.
Diese Methode erzeugt zunächst einige Datenstrukturen für einen Grundstock von Standardklassen wie z.B. `Integer`, `Boolean` oder `Vector` und hängt diese vorübergehend in den abstrakten Syntaxbaum ein. Aufbauend auf diesen Basisdaten wird die in [Plü04a] spezifizierte und in [Ott04] implementierte *Finite Closure* über die Methode `makeFC()` erzeugt. Anschließend werden die oben erwähnten Datenstrukturen für die Standardklassen wieder aus dem Syntaxbaum entfernt.

Bevor der eigentlich TRA mit dem Aufruf von `TRProg()` startet, werden noch einige Hilfsdaten in Form eines Objektes der Klasse `CSupportData` erzeugt und unter anderem mit der Finite Closure initialisiert. Dieses Objekt kapselt für den TRA notwendige Informationen über die Struktur des vorliegenden Programmes und wird beim rekursiven Abstieg in die Unteralgorithmen weitergegeben.

Der eigentliche Ablauf von `typeReconstruction()` besteht aus einer simplen Schleife, die für jede Klasse die Methode `Class.TRProg()` aufruft, deren Rückgabewert am Ende der Menge aller möglichen Typkombinationen aus → Kapitel 4.2.7 entspricht.

4.8.2 Der Unteralgorithmus TRProg

Die Hauptaufgabe von `TRProg` besteht in der Erzeugung der Ausgangsmengen von `CTypeAssumptions` in Form von Instanzen der von `CHashtableSet` abgeleiteten Klasse `CTypeAssumptionSet`. TRProg erzeugt für jede Feldvariable und Methode der aktuellen Klasse eine `CInstVarTypeAssumption` respektive `CMethodTypeAssumption` in das `CTypeAssumptionSet` $V_{fields\_methods}$. Außerdem wird für jede Methode ein `CTypeAssumptionSet` V_i angelegt, das für jeden Methodenparameter dieser Methode eine `CLocalVarTypeAssumption` enthält.

Mit dieser Wissensbasis an Typinformationen wird dann der Algorithmus `TRStart` gerufen, der eine Menge von `CReconstructionTuple` (σ, V) zurückliefert. Jedes `CReconstructionTuple` repräsentiert dabei eine mögliche Typkombination und besteht aus einer Menge von Typannahmen in Form eines `CTypeAssumptionSet` V sowie einer Menge von zugehörigen Substi-

tutionen in Form eines `CSubstitutionSet` σ. Diese Tuple werden mit der Ergebnismenge für die vorherigen Klassen verknüpft und in `CTypeReconstructionResults` verpackt, wobei die `CLocalVarTypeAssumptions` des jeweiligen V_i wieder entfernt werden.

4.8.3 Der Unteralgorithmus TRStart

TRStart bekommt als Argumente n `CTypeAssumptionSets` V_i übergeben; für jede der n Methoden eines. Für jede der n Methoden wird dann der Algorithmus `TRNextMeth` aufgerufen, der ein `CTripleSet` zurückgibt, das m Objekte des Typs `CTriple` enthält. Die Klasse `CTriple` steht genau wie `CReconstructionTuple` für eine mögliche Typkombination mit dem einzigen Unterschied, dass sie zusätzlich noch den berechneten ReturnType des aktuell betrachteten Blocks enthält.
TRNextMeth gibt also für jede der n Methoden m mögliche Typkombinationen der Form (σ_j, ty_j, V_j) zurück. Für jedes ty_j dieser m Möglichkeiten wird daraufhin `TUnify` gerufen, um den ermittelten ReturnType des aktuellen Methodenblocks mit dem ermittelten ReturnType der aktuellen Methode zu unifizieren. Das Ergebnis von `TUnify` sind k mögliche Unifier $unify_j$.

Auf jedes V_j der m möglichen Typkombinationen wird nun jeder der k Unifier $unify_j$ angewendet und eine Menge von `CTupel` (σ_z, V_z) gebildet. Diese Mengen von Tupeln aller V_j werden zu einer Ergebnismenge vereinigt.

Dieser ganze Ablauf wird für jede der n Methoden m Mal ausgeführt, wobei die Erkenntnisse der vorhergehenden Durchläufe immer als Eingabewert für den nächsten Durchlauf dienen. Nach dem letzten Durchlauf wird die Ergebnismenge von Tupeln (σ, V) an `TRProg` zurückgegeben.

Allein anhand dieser Methode wird deutlich wie komplex der TRA sich auf Grund der Kombination unterschiedlicher Typannahmen gestaltet, was sich auch in der Implementierung `Class.TRStart()` in Form von zahlreichen Schleifen wiederfindet.

4.8.4 Der Unteralgorithmus TRNextMeth

`TRNextMeth` hat im Prinzip die Aufgabe, die bisherigen Teilergebnisse repräsentiert durch ein `CReconstructionTupleSet` um weitere Informationen aus der aktuellen Methode zu erweitern.
Dabei werden für alle n V_i der übergebenen Tupel (σ_i, V_i) der Algorithmus `TRStatement` des aktuellen Methodenblocks aufgerufen, der daraufhin `TRStatements` auf seine Statement-Liste ruft. Zuvor wird das jeweilige V_i noch um die Typannahmen der neuen Methodenparameter als `CLocalVarTypeAssumptions` erweitert. Dabei werden die bisher ermittelten Unifier berücksichtigt, d.h. entsprechende Substitutionen gleich angewendet. Im Gegensatz zur Spezifikation in [Plü04a] werden die `CLocalVarTypeAssumptions` der vorigen Methode nicht gelöscht, sondern bleiben erhalten.

4.8.5 Der Unteralgorithmus TRStatements

Der Algorithmus `TRStatements`, der von der Methode `Block.TRStatements()` implementiert wird, hat die Aufgabe, alle Anweisungen eines Blocks auf ihren Typ hin zu untersuchen, sowie den Rückgabetyp des betreffenden Blocks zu bestimmen. Eingabedaten sind das bis zu diesem Zeitpunkt ermittelte `CSubstitutionSet` σ und das `CTypeAssumptionSet` V.

`TRStatements` ruft für die erste Anweisung des Blocks den Unteralgorithmus `TRStatement`, der eine Menge von Typkombinationen in Form von `CTriple` (σ_i, ty_i, V_i) zurückliefert. Daraufhin wird für jedes Triple mit $ty = void$ rekursiv `TRStatement` der nächsten Anweisung des Blocks gerufen und dessen Ergebnismengen vereinigt. So steigt der Algorithmus rekursiv ab, bis die letzte Anweisung des Blocks erreicht ist. Von der Ergebnismenge von `TRStatement` der letzten Anweisung werden alle Triple zurückgeliefert und der Algorithmus steigt wieder auf.

Von den Ergebnis-Triplen der vorigen Anweisungen werden deshalb nur diejenigen genommen, die den Rückgabetyp `Void` haben, weil alles andere nicht einem korrekten Java-Programm entspräche. Nur die letzte Anweisung eines Blocks darf ein Return-Statement sein und somit einen Rückgabetyp besitzen, alle vorigen Anweisungen dürfen kein Return sein und liefern folglich den Typ `Void` zurück.

4.8.6 Die Unteralgorithmen TRStatement und TRExp

Der Unteralgorithmus `TRStatement` hat als Eingabedaten die bisher ermittelte Menge an Substitutionen σ und die Menge an Typannahmen V. Er rekonstruiert die Typen einer einzelnen Anweisung durch rekursive Aufrufe von `TRStatement` seiner Unteranweisungen (z.B. bei einem Block) oder von `TRExp` seiner Ausdrücke (z.B. bei einer Zuweisung).
`TRExp` hat ebenfalls die bisher berechneten σ und V als Eingabe und rekonstruiert die Typen für einen Ausdruck.
Beide Algorithmen haben als Ausgabe die Menge der bisher möglichen Typkombinationen als `CTripleSet`.

Beispiel.:
Der Algorithmus `TRStatement` für die Zuweisung `number = 5+2;` ermittelt mit Hilfe von `TRExp` für den linken Ausdruck `number` den Typ `B` und durch einen weiteren Aufruf von `TRExp` für den rechten Ausdruck `5+2` den Typ `Integer`. Da der linke und der rechte Typterm anschließend miteinander unifiziert werden können, ist die Anweisung korrekt. Es wurde die Typinformation $B \rightarrow Integer$ gewonnen. Die gesamte Anweisung hat den Rückgabetyp `Void`. Wäre `number` beispielsweise vom Typ `Matrix` gewesen, hätten linke und rechte Seite nicht unifiziert werden können und es wäre eine Exception geworfen worden. Ein `Integer` kann nicht einer Variable vom Typ `Matrix` zugewiesen werden.

Das Typematching der in [Plü04a] angegebenen SML-ähnlichen Pseudo-Syntax für die beiden Algorithmen[1] wird in der objektorientierten Implementierung durch Polymorphie realisiert.

[1] siehe Alg. 5.20 bis 5.27 und Alg. 5.30 bis 5.54 aus [Plü04a] in → Anhang C

So ist die abstrakte Methode `TRStatement()` in der Klasse `Statement` und die abstrakte Methode `TRExp()` in der Klasse `Expr` definiert. Die jeweiligen Subklassen überschreiben die Methoden gemäß der spezifizierten Algorithmen.

4.8.7 Die Unteralgorithmen TRTuple und TRMultiply

Diese beiden, in der Klasse `Expr` implementierten Unteralgorithmen haben hauptsächlich die Aufgabe, für den TRExp-Algorithmus des Methodenaufrufs (Alg. 5.33) die Typen der Methodenargumente zu berechnen und alle Typkombinationen zwischen aktuellen und formalen Parametern der jeweiligen Methode zu erzeugen. Mit diesen Typkombinationen wird dann der Algorithmus `TRMCallApp` gerufen, der dann die Typen des Methodenaufrufs rekonstruiert.

4.9 Zusammenfassung des Programmablaufs

Im Folgenden soll der Inhalt der letzten Abschnitte noch einmal kurz in Form einer Pseudo-Anweisungsliste, die den Programmablauf verdeutlichen soll, zusammengefasst werden.

TypeReconstruction()

- Erzeuge Wissensbasis über Standardklassen
- Rufe madeFC() und erzeuge Finite Closure
- Erzeuge SupportData
- Für jede Klasse in Klassenvektor:
  - Rufe ⇒TRProg(SupportData)
- Gib das letzte TypeReconstructionResult zurück

TRProg(SupportData)

- Erzeuge Basistypannahmen $V_{fields\_methods}$ und V_i
- Rufe ⇒TRStart(V_i, $V_{fields\_methods}$, SupportData)
- Verknüpfe alle Tupel mit Ergebnismenge der vorigen Klassen
- Verpacke Tupel in TypeReconstructionResults
- Gib Ergebnismenge zurück

TRStart(V_i, $V_{fields\_methods}$, SupportData)

- $ret_0 = V_{fields\_methods}$

- Für alle Methoden:
  - Rufe ⇒TRNextMeth(V_i, ret_{i-1}, $method_i$, SupportData)
  - Für alle Triple:
    - Unifiziere ReturnTypes
    - Für alle Unifier:
      - Wende Unifier an
      - Füge Ergebnistupel zu Ergebnismenge ret_i hinzu
  - Gehe mit ret_i in nächsten Schleifendurchlauf
- Gib Ergebnismenge zurück

TRNextMeth(V_{next}, (σ_i, V_i), nextMeth, SupportData)

- Für alle Tupel:
  - $\widetilde{V_i} = V_i$ verknüpft mit V_{next}
  - Rufe ⇒TRStatements(σ_i, $\widetilde{V_i}$, SupportData) von nextMeth.Block
  - Vereinige alle Triple mit Ergebnismenge
- Gib Ergebnismenge zurück

TRStatements(σ_i, V_i, k, SupportData)

- Rufe für k-te Anweisung in der Anweisungsliste TRStatement(σ_i, V_i, SupportData) auf
- Wenn k == letzter Index der Anweisungsliste, dann gib Ergebnismenge zurück
- Ansonsten für alle Triple:
  - Rufe rekursiv ⇒TRStatements(σ_j, V_j, $k+1$, SupportData)
  - Vereinige alle Triple mit Rückgabetyp Void der Teilergebnismengen zur Ergebnismenge
- Gib Ergebnismenge zurück

4.10 Abweichungen von [Plü04a]

4.10.1 Lokale Variablen

In → Kapitel 4.4 wurde ausführlich beschrieben, wie das Prinzip der vorliegenden Implementierung auf einzelnen, ausführbaren Typsubstitutionen beruht. Dieser Ansatz und die Tatsache,

dass im Bytecode auch bei lokalen Variablen der Typ angegeben werden muss, macht es unumgänglich, auch die berechneten Typinformationen von lokalen Variablen in die Ergebnismenge aufzunehmen.
[Plü04a] konzentriert sich auf die Berechnung von Typen für Felddeklarationen und verwendet Typannahmen für lokale Variablen nur temporär. Die Typannahmen für lokale Variablen werden nach ihrer Verwendung unter anderem in den Algorithmen `TRNextMeth` (Alg. 5.18a) und `Block.TRStatement` (Alg. 5.19) wieder entfernt. An den betreffenden Stellen weicht deshalb die Implementierung von der Spezifikation ab.

4.10.2 TRStatement für If-Anweisung

Der Algorithmus `If.TRStatement` (Alg. 5.21) geht davon aus, dass entweder sowohl der then-Block als auch der else-Block einer Verzweigung eine `return`-Anweisung enthalten oder beide den Rückgabewert `void` haben. Die Möglichkeit, dass nur einer der beiden Zweige ein `return` enthält, wird nicht in Betracht gezogen. Gemäß des Sprachumfangs der Programmiersprache Java erweitert die vorliegende Implementierung die Spezifikation des Algorithmus und lässt auch diesen Fall zu.

4.10.3 TRExp für Operatoren

Die zu Projektbeginn vorliegende Version von [Plü04a] ist in Bezug auf den Großteil der `TRExp`-Algorithmen für Operatoren (Alg. 5.41, 5.42 und 5.44 bis 5.52) unvollständig bzw. fehlerhaft.

So werden in diesen Algorithmen die Typen der Operanden berechnet und nur diejenigen Ergebnis-Triple weiter verwertet, die vom Typ `Integer` respektive `Boolean` sind. Alle anderen Typen werden weggeworfen bzw. führen zu einer Fehlermeldung. Dieser Ansatz erscheint auf den ersten Blick logisch, da beim UND-Operator beispielsweise beide Operanden `Boolean` sein müssen. Aus Sicht des Typinferenzsystems ist dies jedoch völlig unzureichend, da die Algorithmen nach dieser Spezifikation ausschließlich eine Typüberprüfung vornehmen, jedoch keinerlei neue Typerkenntnisse erzeugen.

Die Algorithmen müssen so angepasst werden, dass die Typen der Operanden mit `Integer` repsektive `Boolean` über `TUnify` unifiziert werden. Über die Unifizierung wird nicht nur eine Plausibilitätsprüfung durchgeführt, sondern auch neue, zusätzliche Erkenntnisse über die Operandentypen erzeugt. Die Implementierung folgt diesem Ansatz und weicht hier von [Plü04a], das in Kürze wohl überarbeitet wird, ab.

Kapitel 5

Schlussbetrachtungen

5.1 Zusammenfassung

Im Rahmen dieseser Studienarbeit wurde der in [Plü04a] spezifizierte Typinferenzalgorithmus, so weit es der zu Projektbeginn vorliegende Compiler zuließ, nahezu vollständig implementiert.

Von den 40 beschriebenen Algorithmen 5.17 bis 5.57 wurden außer den Unteralgorithmen 5.32, 5.39 und 5.55 alle umgesetzt. Die ersten beiden Algorithmen beziehen sich auf Arrays, die vom vorliegenden Compiler nicht unterstützt werden. Der Algorithmus 5.55 `TUnify` war bereits in der Vorgängerstudienarbeit [Ott04] realisiert worden.

Der Typinferenzalgorithmus scheint fehlerfrei zu funktionieren. Die Typen für das in [Plü04a] beschriebene Matrix-Usecase-Programm (→ Anhang B) wurden korrekt berechnet und die typlosen Variablen folgerichtig substituiert.

5.2 Ausblick

Obwohl der Compiler mittlerweile recht stabil erscheint, besitzt er noch immer einige kleinere Fehler und deckt auch nicht den gesamten Sprachumfang von Java ab.
So werden weder Arrays noch BaseTypes unterstützt. Im Rahmen dieser Arbeit wurde der Parser zwar so angepasst, dass er BaseTypes erzeugt, aber weder `TUnify` noch die Bytecode-Generierung unterstützen diese. Aus diesem Grund mussten auch im Typrekonstruktionsalgorithmus Workarounds eingebaut und statt den BaseTypes `IntegerType`, `BooleanType` und `CharacterType` RefTypes mit ensprechenden Namen verwendet werden.

Was die Typinferenz angeht, so sind auch in diesem Bereich noch einige Korrekturen und Verbesserungen vorzunehmen. So funktioniert der Typrekonstruktionsalgorithmus bislang nur für eine einzelne Klasse. Die Kombination von Typen über mehrere selbst definierte Klassen hinweg per Funktionsaufruf wurde nicht berücksichtigt. Sollte dies realisiert werden, müssten einige Modifikationen am Algorithmus vorgenommen und insbesondere ein komplexerer Intersection Type nach [Plü04a] implementiert werden.

Des Weiteren ist der Parser noch nicht in der Lage, im Quellcode auftauchende generische Typvariablen von gewöhnlichen RefTypes zu unterscheiden. Auch `TUnify` und der Typrekonstruktionsalgorithmus arbeiten bisher nur mit Instanzen von `TyploseVariable` und unterstützen weder die Verarbeitung von `GenericTypeVar`-Instanzen noch den Rückschluss von typlosen Variablen auf generische Typparameter einer Klasse. Somit werden Programme mit generischen Klassen letztlich nur bedingt unterstützt.

Ebenfalls verbesserungswürdig ist die Bytecode-Generierung, die teilweise fehlerhaft zu sein scheint und einmal gründlich überarbeitet werden sollte. In diesem Zusammenhang müsste auch die fehlende Type Erasure (→ Kapitel 2.3.2) implementiert werden.

Anhang A

Einrichten der Entwicklungsumgebung

A.1 Installation von Eclipse

Vorraussetzung für die Eclipse-Installation ist ein korrekt eingerichtetes Sun JDK 1.5, das von www.java.sun.com heruntergeladen werden kann. Ist ein solches vorhanden, kann die neueste Version von Eclipse unter www.eclipse.org als zip-Datei heruntergeladen und entpackt werden.

A.2 Einrichten von CVS

Nach dem Starten von Eclipse kann über die Perspektive „CVS Repository Exploring“ eine neue „Repository Location“ angelegt werden.

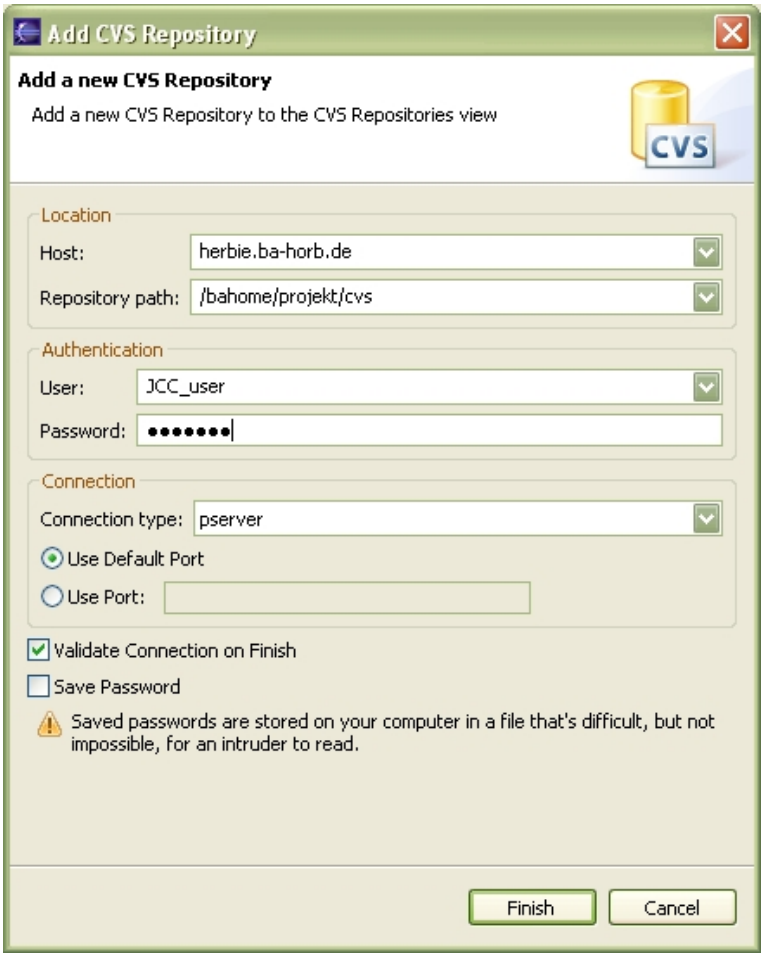

Abbildung A.1: Repository Location Dialog

Das aktuelle Java Compiler Projekt befindet sich im CVS-Repository der BA Horb im Modul „JavaCompilerCore“, das Projekt der Partnerstudienarbeit im Modul „JavaCompilerPlugin“. Die Zugangsdaten sehen wie folgt aus:

Modul	JavaCompilerCore	JavaCompilerPlugin
Host	herbie.ba-horb.de	herbie.ba-horb.de
Repository	/bahome/projekt/cvs	/bahome/projekt/cvs
Verbindungsart	pserver	pserver
User	JCC_user	JCP_user
Passwort	jcc_36y	jcp_19z

Die Passwörter können per Anfrage ans Rechenzentrum zurückgesetzt werden.

A.3 Auschecken und Konfigurieren des Projektes

Über die CVS-Perspektive kann das Repository der BA durchsucht und das Compiler Projekt per rechte Maustaste und Auswahl des Menüpunktes `Checkout` in den lokalen Eclipse-Workspace geladen werden.

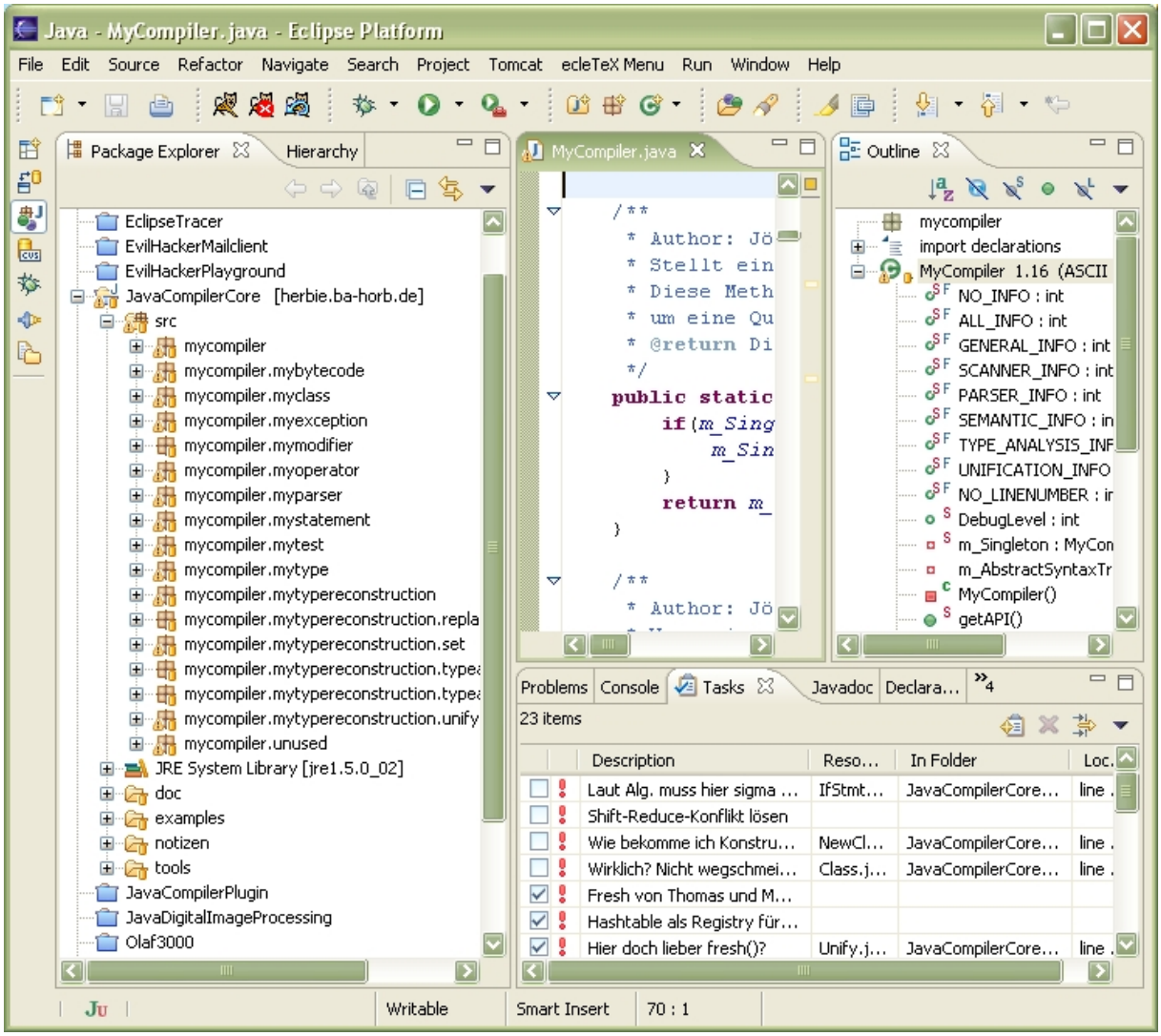

Abbildung A.2: Java Compiler Projekt

Im Unterverzeichnis `/tools/cygwin` findet sich die Setup-Datei und die Installationsdateien, um eine Cygwin-Umgebung unter Windows einzurichten.

Abschließend kann über das Menü „Project>Properties>Builders“ der JavaParserBuilder ausgewählt und über den Button „Edit...“ ein Dialog zur Auswahl des betriebssystemabhängigen Ant-Scripts geöffnet werden. Die Scripts hierfür befinden sich im Unterverzeichnis `/tools` des Projektes.

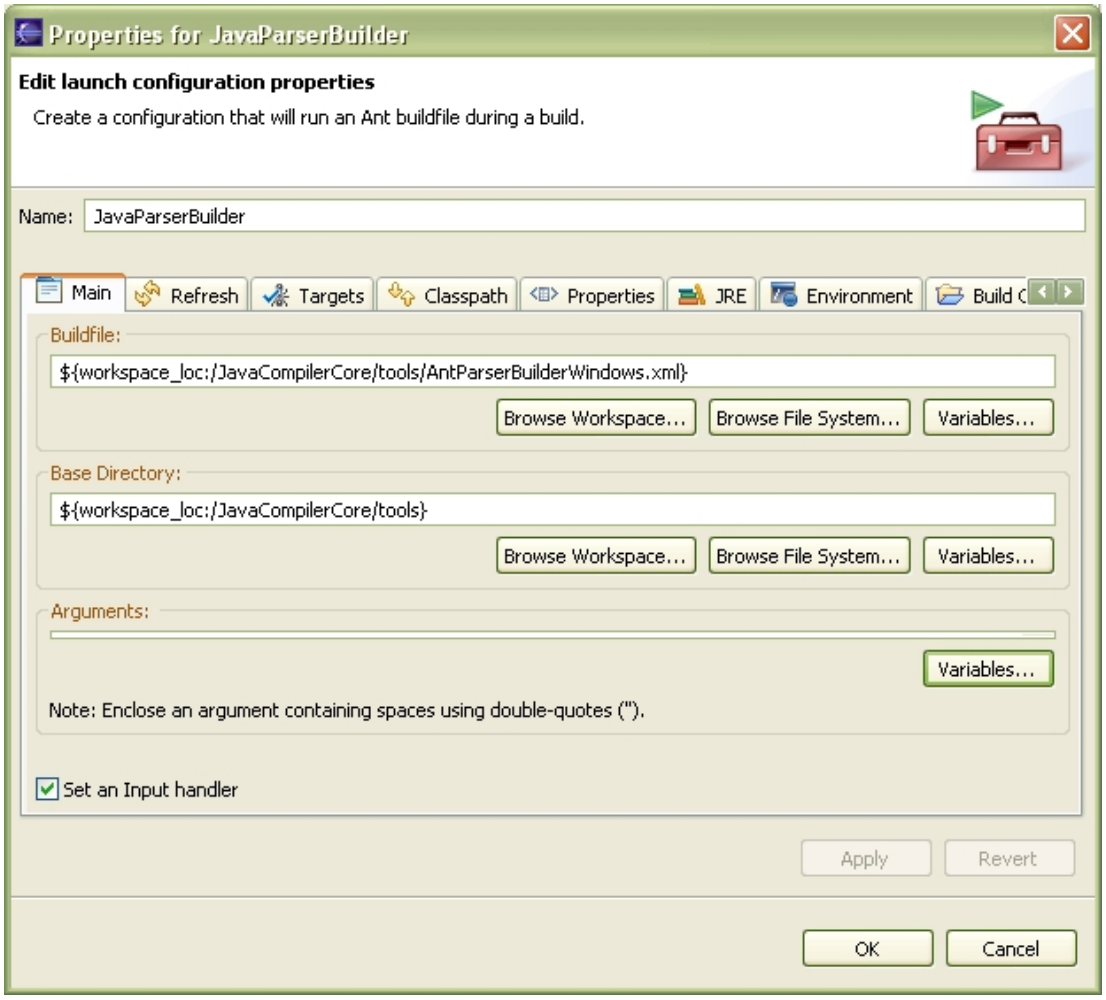

Abbildung A.3: Ant Builder Dialog

Anhang B

Usecase-Programm

```
01:  class Matrix extends Vector<Vector<Integer>> {
02:
03:     mul(m){
04:        ret = new Matrix();
05:        i = 0;
06:        while(i <size()) {
07:           v1 = this.elementAt(i);
08:           v2 = new Vector<Integer>();
09:           j = 0;
10:           while(j < v1.size()) {
11:              erg = 0;
12:              k = 0;
13:              while(k < v1.size()) {
14:                 erg = erg + v1.elementAt(k).intValue()
                                * m.elmentAt(k).elementAt(j).intValue();
15:                 k++;
16:              }
17:              v2.addElement(new Integer(erg));
18:              j++;
19:           }
20:           ret.addElement(v2);
21:           i++;
22:        }
23:        return ret;
24:     }
25:
26:  }
```

Anhang C

Plümickesche Algorithmen

Type Reconstruction

Before we present the type reconstruction algorithm we give its main data structures:
The set of type assumption A is, similar to the set of type assumption of the type inference rules, a family, which maps class names (types) to the family of its typed fields and method names. In difference to the family of type assumptions of the inference rules, now intersection types are needed. For technical reasons, we map each method and its number of arguments (as its index) to an intersection type:

$$\begin{array}{rl}(m^{(n)}: & (\forall\alpha_{1,1}\ldots(\forall\alpha_{1,p_1}(\theta_{1,1}\times\ldots\times\theta_{1,n}\rightarrow\theta_1))\ldots)\\ & \wedge\ldots\wedge\\ & (\forall\alpha_{o,1}\ldots(\forall\alpha_{o,p_o}(\theta_{o,1}\times\ldots\times\theta_{o,n}\rightarrow\theta_o))\ldots)\in A_\theta\end{array}$$

where $A=(A_\theta)_{\theta \text{ is a class}}$
Let $T_{TC}(\,BTV\,)$ be the set of type terms over the type signature $(BTV,\ TC)$ on declared G–JAVA classes and $\leq$ a type term ordering on $T_{TC}(\,BTV\,)$. Furthermore we need $\mathbf{FC}(\leq)$. The G–JAVA class which methods should be typed is given as an abstract syntax tree (cp. appendix **??**) of the following form:

class(ClassName(*clType*), extends(*supertype*),
InstVarDecl(x_1,θ_1) , . . . , InstVarDecl(x_o,θ_o)
Method(m_1,
$rety_1$,
$(v_{1,1}:ty_{1,1}\ldots v_{1,m_1}:ty_{1,m_1})$
Block(B_1))
. . .
Method(m_n,
$rety_n$,
$(v_{n,1}:ty_{n,1}\ldots v_{n,m_n}:ty_{n,m_n})$
Block(B_n)))

In the following we denote sets of type assumption as $A_{(f,n)}$ which is an abbreviation for the

set of type assumptions A excluding the type assumptions for the function symbols f with n argument positions.
During the type reconstruction for each assumed type of the intersection type of a method there is a triple

$$(\sigma, \theta, V)$$

generated, where

- σ is a substitution which maps type variables of the originally assumed types to the types which are reconstructed for the type variables
- θ stands for the result type of the actual expression or statement list.
- V is set of type assumptions of
  - the actual method (for recursive calls)
  - the fields of the actual class
  - local variables

The following variables are considered as global in the whole type reconstruction algorithm.

- act_cl contains the type $clType$ of actual class *jclass*
- the type term ordering of the declared G–JAVA classes: $\leq$
- the finite closure $\mathbf{FC}(\leq)$.
- A is the set of type assumptions for the already reconstructed type, indexed by the class-names.

Algorithm C.0.17 (TRprog) The algorithm is the main procedure of the type reconstruction which calls the different sub-functions.

$$
\begin{array}{l}
\mathsf{TRprog}(\, A, jclass\,) = \\
\quad \textbf{let} \\
\qquad \mathsf{NewTVar}(\, jclass\,) = \mathsf{class}(\, \mathsf{ClassName}(\, clType, \mathsf{extends}(\, supertype\,)\ ), \\
\qquad\qquad\qquad\qquad \mathsf{InstVarDecl}(\, x_1, \theta_1\,)\ , \ldots, \mathsf{InstVarDecl}(\, x_o, \theta_o\,) \\
\qquad\qquad\qquad\qquad \mathsf{Method}(\, m_1,\ rety_1,\ (v_{1,1} : ty_{1,1} \ldots v_{1,m_1} : ty_{1,m_1}), Bl_1\,) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad \mathsf{Method}(\, m_n,\ rety_n,\ (v_{n,1} : ty_{n,1} \ldots v_{n,m_n} : ty_{n,m_n}), Bl_n\,)\ \ ) \\
\qquad \textbf{(1)}\ V_{fields\_methods} = \{\, \mathsf{this}.x_1 : \theta_1, \ldots, \mathsf{this}.x_o : \theta_o \,\} \\
\qquad\qquad\qquad\qquad \cup \{\, me_1 : ty_{1,1} \times \ldots \times ty_{1,m_1} \rightarrow rety_1 \,\} \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad \cup \{\, me_n : ty_{n,1} \times \ldots \times ty_{n,m_n} \rightarrow rety_n \,\}
\end{array}
$$

(2) $V_1 = \{\, v_{1,1} : ty_{1,1} \ldots v_{1,m_1} : ty_{1,m_1} \,\}$
$\vdots$
$V_n = \{\, v_{n,1} : ty_{n,1} \ldots v_{n,m_n} : ty_{n,m_n} \,\}$
(3) $\{\, (\sigma_1, \overline{V}_1), \ldots, (\sigma_l, \overline{V}_l) \,\} = \mathsf{TRstart}(\, (Bl_1, \ldots, Bl_n), (V_1, \ldots, V_n), V_{fields\_methods} \,)$
in
let
$A_{clType} = \mathsf{intersec}\ (\, A \cup \mathsf{clear}(\, \bigcup_{1 \leqslant i \leqslant l} \mathbf{bind}(\, \mathsf{TVar}(\, clType \,), \overline{V}_i \,) \,) \,)$
in
$A \cup A_{clType}$
end
end

The algorithm TRprog first determines by the funtcion NewTVar new type variables as types for parameters and return types of the methods where the type is not explicitly given. In explicitly given types the containing type variables are considered as type constants, because it is not allowed to substitute these type variables during the type reconstruction.
The set of type assumptions $V_{fields\_methods}$ contains the type assumptions for all methods of the class *jclass*. The sets $V_1, \ldots, V_n$ contain the type assumptions for the parameters of the respective methods.
The function TRstart calls the type reconstruction of the blocks of the respective methods. The results are sets of pairs, which contain a type unifier σ_i and a set of type assumptions, with the reconstructed types. The type unifier is no longer needed at this point, because they have been already used to reconstruct the type assumptions.
The function clear removes the assumptions for the parameters and the local variables.
Finally, the intersection types of the reconstructed types of the methods are built (intersec).

Sub-functions of the type reconstruction

The type reconstruction is started by the function TRstart.

Algorithm C.0.18 (TRstart) The function TRstart controls the type reconstruction of the different methods.

$\mathsf{TRstart}(\, (Bl_1, \ldots, Bl_n), (\overline{V}_1, \ldots, \overline{V}_n), V_{fields\_methods} \,) =$
let
$\overline{V}_0 = \emptyset$
$ret_0 = \{\, ([\,], V_{fields\_methods}) \,\}$
let-foreach $1 \leqslant i \leqslant n$:
$ret_i =$
let
$\{\, (\sigma_1, ty_1, V_1), \ldots, (\sigma_m, ty_m, V_m) \,\} = \mathsf{TRNextMeth}(\, \overline{V}_{i-1}, \overline{V}_i, ret_{i-1}, Bl_i \,)$
let-foreach $1 \leqslant j \leqslant m$:
$\theta_j = \mathsf{RetType}(\, me_i, V_j \,)$

$$
\begin{array}{l}
\quad\quad unify_j = \mathbf{TUnify}_{\leq^*}(\, ty_j, \theta_j \,) \\
\quad\; \mathbf{in} \\
\quad\quad \bigcup_{1 \leqslant j \leqslant m} \{\, unify, \mathbf{sub}(\, unify, V_j \,) \mid unify \in unify_j \,\} \\
\quad\; \mathbf{end} \\
\mathbf{in} \\
\quad ret_n \\
\mathbf{end}
\end{array}
$$

The types of the blocks, which determine the respective methods are reconstructed step by step by the function TRNextMeth. During this reconstruction the types are adapted. After the type reconstruction of a block, the return type of the block and the return type of the method are unified. Finally, for each reconstructed triple (type unifier, return type, set of adpted type assumptions) of the previous method the types of the next method are reconstructed by the function TRNextMeth.

$$
\begin{array}{l}
\mathsf{TRNextMeth}(\, \overline{V}_{last}, \overline{V}_{next}, \{\, (\sigma_1, V_1), \ldots, (\sigma_n, V_n) \,\}, Bl \,) = \\
\quad \bigcup_{1 \leqslant i \leqslant n} \mathsf{TRStmt}(\, \sigma_i, V_i \backslash \{\, v : ty \mid (v : ty') \in \overline{V}_{last} \,\} \cup \mathbf{sub}(\, \sigma_i, \overline{V}_{next} \,), Bl \,)
\end{array}
$$

The RetType determines the return type of the given method in a set type assumptions.

$$
\begin{array}{l}
\mathsf{RetType}(\, me_i, V \,) = \\
\quad \mathbf{let} \\
\quad\quad (me_i : \omega_1 \times \ldots \times \omega_n \rightarrow \theta) \in V \\
\quad \mathbf{in} \\
\quad\quad \theta \\
\quad \mathbf{end}
\end{array}
$$

In TRNextMeth the function TRStmt is called, which determines the type of a statement.

TRStmts

Now we describe the type reconstruction algorithm of the statements in different functions. The goal of these functions is to determine the result type of the corresponding method. In principle we have to differ two cases. Either a method has no return type, which is described by `void` or it has a type term as result type. In G–JAVA an existing result type is determined by a return statement, which stands always at the end of a list of statements. The type reconstruction algorithm make allowance for this by first assuming that the result type is `void`. If finally a return statement is given, the type of the respective expression is determined and the result type of the respective method is identified.
The structure of the statements in the abstract syntax is given such that for each statement there is one construct. A list of statements is subsumed by the Block-construct. From this construction follows the structure of the function TRStmts respectively TRStmt. The function

TRStmts takes a list of statements and calls the function TRStmt for each statement. For each statement construct there is a function TRStmt which reconstructs the types for the respective statement.

Algorithm C.0.19 (TRStmts) The function TRStmts calls for each element of a list of statements the function TRStmt.

$\mathsf{TRStmts}(\,\overline{\sigma}, V, s :: [], V_{start}\,) =$
  let
    $\{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRStmt}(\,\overline{\sigma}, V, s\,)$
  let-foreach $1 \leqslant i \leqslant n$:
    $V_i' = \{\, f : ty' \mid (f : ty') \in V_i \wedge \exists ty \in \mathsf{Type}(\,T_{TC}(\,BTV\,)\,) : (f : ty) \in V_{start}\,\}$
  in
    $\{\,(\sigma_1, ty_1, V_1'), \ldots, (\sigma_n, ty_n, V_n')\,\}$
  end

In the case of the last element of the list of statements the local variables, which are added during the type reconstruction of the block are removed.

$\mathsf{TRStmts}(\,\overline{\sigma}, V, s :: stmts, V_{start}\,) =$
  let
    $\{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRStmt}(\,\overline{\sigma}, V, s\,)$
  in
    $\bigcup_{\substack{1 \leqslant i \leqslant n \\ ty_i = \mathtt{void}}} \mathsf{TRStmts}(\,\sigma_i, V_i, stmts, V_{start}\,)$
  end

In the other case, that the statement is not the last statement of a statement list, the function TRStmt is called. With the result the function TRStmts is called recursively for the rest of the statement list.

In the following we consider the function TRStmt. We give for each statement an own algorithm, which determines the types of the respective statement. The different algorithms call each other mutually recursive.

Algorithm C.0.20 (TRStmt for a block) The function TRStmt for a Block removes the Block–construct and calls the function TRStmts for the including statement list.

$\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{Block}(\,B\,)\,) = \mathsf{TRStmts}(\,\overline{\sigma}, V, B, V\,)$

Algorithm C.0.21 (TRStmt for an if–statement) The function TRStmt for the if–statement reconstructs first the types for the conditional expression, second the types for the *then*–branch and last the types for the *else*–branch. If a result types of the *then*– respectively the *else*–branch exist these types are unified.
$\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{If}(\,e, \mathsf{Block}(\,B_1\,), \mathsf{Block}(\,B_2\,)\,)\,) =$
  let

$\{ (\sigma'_1, \texttt{boolean}, V'_1), \ldots, (\sigma'_m, \texttt{boolean}, V'_m) \} = \mathsf{TRExp}(\, \overline{\sigma}, V, e \,)$
$\{ (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \} = \bigcup_{1 \leqslant k \leqslant m} \mathsf{TRStmts}(\, \sigma'_k, V'_k, B_1, V'_k \,)$

in
let-foreach $1 \leqslant i \leqslant n$:
$\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,o_i}, ty_{i,o_i}, V_{i,o_i}) \} = \mathsf{TRStmts}(\, \sigma_i, V_i, B_2, V_i \,)$
if $ty_1 \neq \texttt{void}$ **then**
let-foreach $1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant o_i, ty_i \neq \texttt{void}, ty_{i,j} \neq \texttt{void}$:
$unify^1_{i,j} = \mathbf{TUnify}_{\leq^*}(\, ty_{i,j}, ty_i \,)$
$unify^2_{i,j} = \mathbf{TUnify}_{\leq^*}(\, ty_i, ty_{i,j} \,)$
in
$\bigcup_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant o_i}} \{ (\sigma, \sigma(\, ty_i \,), \mathbf{sub}(\, \sigma, V_{i,j} \,)) \mid \sigma \in unify^1_{i,j} \} \cup$
$\bigcup_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant o_i}} \{ (\sigma, \sigma(\, ty_{i,j} \,), \mathbf{sub}(\, \sigma, V_{i,j} \,)) \mid \sigma \in unify^2_{i,j} \}$
end
else
if $ty_{1,1} = \texttt{void}$ **then**
$\{ ([\,], ty_{i,1}, V_{i,1}), \ldots, ([\,], ty_{i,o_i}, V_{i,o_i}) \}$
else
$\emptyset$
end

The type reconstruction of the conditional expression must have the result types `boolean`. All other reconstructed types are not considered.
With the possibly different result types of the conditional expression, the types of the *then*–branch are determined.
Foreach result of this reconstruction the types of the *else*–branch are reconstructed. It is necessary to know which reconstructed type of the *then*–branch belongs to which reconstructed type of the *else*–branch, as these types are unified for the result type of the If–statement.
If the *then*– and the *else*–branch have no result type (in the algorithm described as the type `void`) the result type of the If–statement is also `void`. It is enough to consider one type to decide if the the type is `void`, as the type `void` is determined by the not existing of a Return–statement in each branch, respectively.

Algorithm C.0.22 (TRStmt for an Return–statement) The result type of an Return–statement is determined by the type of the returned expression. Because of this the function TRExp is called for the expression, which reconstructs the types of the expression.

$\mathsf{TRStmt}(\, \overline{\sigma}, V, \mathsf{Return}(\, e \,) \,) = \mathsf{TRExp}(\, \overline{\sigma}, V, e \,)$

Algorithm C.0.23 (TRStmt for an While–statement) In the function TRStmt for the While–statement first the types for the ending condition are reconstructed. The result type

must be `boolean`. All other reconstructed types are not considered.
Then, the function TRStmts is called for the list of statements, which is included in the corresponding block.

$$\begin{array}{l}\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{While}(\,e, \mathsf{Block}(\,B\,)\,)\,) = \\ \quad \textbf{let} \\ \quad\quad \{\,(\sigma'_1, \texttt{boolean}, V'_1), \ldots, (\sigma'_n, \texttt{boolean}, V'_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e\,) \\ \quad \textbf{in} \\ \quad\quad \bigcup_{1 \leqslant i \leqslant n} \mathsf{TRStmts}(\,\sigma'_i, V'_i, B, V'_i\,) \\ \quad \textbf{end}\end{array}$$

Algorithm C.0.24 (TRStmt for an LocalVarDecl–statement) In the function TRStmt for the LocalVarDecl–construct two cases must be distinguished. In the first case the local variable is declared without type. In the second case the type is also declared.

$$\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{LocalVarDecl}(\,v\,)\,) = \{\,(\overline{\sigma}, \texttt{void}, V \cup \{\,v : \mathsf{fresh}(\,a\,)\,\})\,\}$$

In the first case for the declared variable a fresh type variable is assumed as its type. This typed local variable is added to the set of type assumptions V.

$$\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{LocalVarDecl}(\,v, \theta\,)\,) = \{\,(\overline{\sigma}, \texttt{void}, V \cup \{\,v : \theta\,\})\,\}$$

In the second case the typed local variable is added to the set of type assumptions V.

The next three algorithms reconstruct the types of statements, which are also expressions. In the algorithms the function TRExp is called, which reconstructs the types for the corresponding expressions.

Algorithm C.0.25 (TRStmt for an Assign–statement) In the function TRStmt for the Assign–statement the types of the corresponding Assign–expression are determined. For each correct reconstructed type of the expression the substitutions and the sets of type assumptions are also given as result of the Assign–statement. Only, the result types are changed to `void`.

$$\begin{array}{l}\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{Assign}(\,e_1, e_2\,)\,) = \\ \quad \textbf{let} \\ \quad\quad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Assign}(\,e_1, e_2\,)\,) \\ \quad \textbf{in} \\ \quad\quad \{\,(\sigma_1, \texttt{void}, V_1), \ldots, (\sigma_n, \texttt{void}, V_n)\,\} \\ \quad \textbf{end}\end{array}$$

Algorithm C.0.26 (TRStmt for an New–statement)
$\mathsf{TRStmt}(\,\overline{\sigma}, V, \mathsf{New}(\,\theta, (t_1, \ldots, t_n)\,)\,) =$
let

$\{ (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \} = \mathsf{TRExp}(\overline{\sigma}, V, \mathsf{New}(\theta, (t_1, \ldots, t_n)))$
in
$\{ (\sigma_1, \texttt{void}, V_1), \ldots, (\sigma_n, \texttt{void}, V_n) \}$
end

Algorithm C.0.27 (TRStmt for an MethodCall–statement)
$\mathsf{TRStmt}(\overline{\sigma}, V, \mathsf{MethodCall}(e, f(e_1, \ldots, e_n)))$ =
let
$\{ (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \} = \mathsf{TRExp}(\overline{\sigma}, V, \mathsf{MethodCall}(e, f(e_1, \ldots, e_n)))$
in
$\{ (\sigma_1, \texttt{void}, V_1), \ldots, (\sigma_n, \texttt{void}, V_n) \}$
end

TRExp

First, we give two functions TRtuple and TRmultiply, which are called from other functions to determine different types for a tuple of subterms. The function TRtuple calls the function TRmultiply for each subterm one after another. The function TRmultiply determines the different correct types of the next subterm and multiplies the result triple if there is more than one correct type for the subterm.

Algorithm C.0.28 TRtuple calls step by step for each subterm TRmultiply and unites the results. $\mathsf{TRtuple}(result, (t_1, \ldots, t_n)) =$

let
$\{ result_1, \ldots, result_l \} = \mathsf{TRmultiply}(result, t_1)$
in
if $n \neq 1$ **then**
$\bigcup_{1 \leqslant i \leqslant l} \mathsf{TRtuple}(result_i, (t_2 \ldots t_n))$
else
$\{ result_1, , \ldots, result_l \}$
end

Algorithm C.0.29 TRmultiply determines the different types of the subterms and multiplies the result triples, which represent respectively one type of the tuple of expressions.

$\mathsf{TRmultiply}((\sigma, (\theta_1 \ldots \theta_m), V), t) =$
let
$\{ result_1, \ldots, result_l \} = \mathsf{TRExp}(\sigma, V, t)$
in
$\{ (\sigma', (\sigma'(\theta_1) \ldots \sigma'(\theta_m), \theta'), V') \mid (\sigma', \theta', V') = result_i, 1 \leqslant i \leqslant l \}$
end

Algorithm C.0.30 (TRExp for Assign) In the function first the both expression are considered as a tuple. Their types are reconstructed one after another by the function TRtuple. After that the two result types are unified.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Assign}(\,e_1, e_2\,)\,) =$
let
$\{\,(\sigma_1, (ty_{1,1}, ty_{1,2}), V_1), \ldots, (\sigma_n, (ty_{n,1}, ty_{n,2}), V_n)\,\} = \mathsf{TRtuple}(\,(\overline{\sigma}, \epsilon, V), (e_1, e_2)\,)$
in
let-foreach $1 \leqslant i \leqslant n$:
$unify = \mathbf{TUnify}_{\leq^*}(\,ty_{i,2}, ty_{i,1}\,)$
$substset_i = \{\,(\sigma \circ \sigma_i, \sigma(\,ty_{i,1}\,), \mathbf{sub}(\,\sigma, V\,)) \mid \sigma \in unify\,\}$
in
$(\bigcup_{1 \leqslant i \leqslant n} substset_i)$
end
end

Algorithm C.0.31 (TRExp for the New–operator) The application of the New–operator is equivalent to the application of a method. Therefore as for the method application, the function TRMCallApp is called (algorithm C.0.33). The type of the receiver of the New–operator is given as the type argument of the New–operator itself.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{New}(\,\theta, ()\,)\,) = \{\,(\overline{\sigma}, \theta, V)\,\}$

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{New}(\,\theta, (t_1, \ldots, t_n)\,)\,) =$
let
$\{\,(\overline{\sigma}_1, (\overline{\nu}_1 \ldots \overline{\nu}_n)_1, V_1), \ldots, (\overline{\sigma}_l, (\overline{\nu}_1 \ldots \overline{\nu}_n)_l, V_l\,\} = \mathsf{TRtuple}(\,(\overline{\sigma}, \epsilon, V), (t_1, \ldots, t_n)\,)$
in
$\bigcup_{1 \leqslant i \leqslant l} \mathsf{TRMCallApp}(\,(\overline{\sigma}_i, (\theta(\overline{\nu}_1 \ldots \overline{\nu}_n)_i), V_i, \texttt{<init>}_\theta(\,t_1, \ldots, t_n\,)\,)$
end

Algorithm C.0.32 (TRExp for the NewArray–operator) For a new array an `int`–expression must be evaluated to determine the length of the array. The array type is determined by the argument of the NewArray–operator.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{NewArray}(\,\theta, t\,)\,) =$
let
$(\sigma, ty, V') = \mathsf{TRExp}(\,V, t\,)$
in
if $ty \neq$ `int` **then**

$\emptyset$
else
$\{ (\sigma, \theta[], V') \}$
end

The next algorithm reconstructs types for method applications. This reconstruction is divided in the functions TRExp and TRMCallApp. The function TRExp is the main procedure which starts the recursive type reconstruction by calling TRtuple for the subterms $t_1, \ldots, t_n$.
If for all subterms of a function application the type reconstruction is completed, TRExp calls the function TRMCallApp. In TRMCallApp the type assumptions of the function are unified with the result types of the subterms. In this function the overloading of methods is propagated. If there is more than one unifiable type of a method, another set of type assumptions V for the actual method is generated.

Algorithm C.0.33 (TRExp for method applications) First we give the main procedure TRExp. Here we consider the receiver of the method application as a further argument. Therefore, the function TRtuple is called with the receiver expression and the argument expressions of the method application.

$\mathsf{TRExp}(\overline{\sigma}, V, \mathsf{MethodCall}(re, f(t_1, \ldots, t_n))) =$
let
$\{ result_1, , \ldots, result_l \} = \mathsf{TRtuple}((\overline{\sigma}, \epsilon, V), (re, t_1, \ldots, t_n))$
in
$\bigcup_{1 \leqslant \lhd} \mathsf{TRMCallApp}(result_i, f(t_1, \ldots, t_n))$
end

The following function TRMCallApp determines the types of the application of a method.

Algorithm C.0.34 (TRMCallApp) For the function TRMCallApp let $\overline{\theta}_{j,1} \times \ldots \times \overline{\theta}_{m,j} \rightarrow \overline{\theta}_j$ be the j'th type assumption of the intersection type of the method f in the class $\overline{\theta}_{j,0}$. Then, TRMCallApp type unifies the type assumptions of the class $\overline{\theta}_{j,0}$ and of the arguments $\overline{\theta}_{j,1}, \ldots, \overline{\theta}_{m,j}$ of the method f with the receiver type $\overline{\nu}_0$ and the result types $\overline{\nu}_1, \ldots, \overline{\nu}_n$ of the subterms $t_1, \ldots, t_n$ (cf. figure **??**). For each type assumption of f, where the type unifications do not fail, a new result triple is generated, which represents one type of the actual subterm. The type term $\sigma(\overline{\theta}_j)$ represents the result type of the term $f(t_1, \ldots, t_n)$ and $\mathbf{sub}(\sigma, V)$ consists of the new assumptions for the variables and the methods of the actual class. In the algorithm the not recursive application and the recursive application differ. The result of the not recursive application is denoted by $case_1$ and the result of the recursive application is denoted by $case_2$.

$\mathsf{TRMCallApp}((\overline{\sigma}, (\overline{\nu}_0 \overline{\nu}_1 \ldots \overline{\nu}_m), V), f(t_1, \ldots, t_n)) =$
let

$case_1 =$
let-foreach $A_{\forall a_1.....\forall a_n.C<a_1,...,a_n>}$ with $(f^{(m)} : ty) \in A_{\forall a_1.....\forall a_n.C<a_1,...,a_n>}$:
$A_{\forall a_1.....\forall a_n.C<a_1,...,a_n>} = A_{\forall a_1.....\forall a_n.C<a_1,...,a_n>} \backslash (f^{(m)})$[1]
$\cup \ \{ f^{(m)} : \ (\theta^C_{1,1} \times \ldots \times \theta^C_{1,m} \to \theta^C_1)$
$\wedge \ldots \wedge$
$(\theta^C_{o_C,1} \times \ldots \times \theta^C_{o_C,m}, \to \theta^C_{o_C}) \}$

$res_C =$
let-foreach $1 \leqslant j \leqslant o_C$:
$(\overline{\theta}_{j,0}, \overline{\theta}_{j,1} \times \ldots \times \overline{\theta}_{j,m} \to \overline{\theta}_j) = \mathsf{fresh}(\ (C<a_1, \ldots, a_n>), \theta^C_{j,1} \times \ldots \times \theta^C_{j,m} \to \theta^C_j)$
$unify_j = \mathbf{TUnify}_{\leq^*}(\ (\overline{\nu_0}, \overline{\nu}_1, \ldots, \overline{\nu}_m), (\overline{\theta}_{j,0}, \overline{\theta}_{j,1}, \ldots, \overline{\theta}_{j,m})\ )$
$substset_j = \{\ (\sigma \circ \overline{\sigma}, \sigma(\ \overline{\theta}_j\ ), \mathbf{sub}(\ \sigma, V\ ))\ |\ \sigma \in unify_j\ \}$
in
$(\bigcup_{1 \leqslant j \leqslant o_C} substset_j)$
end
in
$(\bigcup_C ret_C)$
end
$case_2 =$
if $(f^{(m)} : ty) \in V$ **then**
let
$V = V \backslash (f^{(m)}) \cup \{\ f^{(m)} : \theta_1 \times \ldots \times \theta_m \to \theta\ \}$
$unify = \mathbf{TUnify}_{\leq^*}(\ (\overline{\nu}_0 \overline{\nu}_1, \ldots, \overline{\nu}_n), (act\_cl, \theta_1, \ldots, \theta_n)\ )$
in
$\{\ ((\sigma \circ \overline{\sigma}), \sigma(\ \theta\ ), \mathbf{sub}(\ \sigma, V\ ))\ |\ \sigma \in unify\}$
end
else
$\emptyset$
in
$case_1 \cup case_2$
end

Algorithm C.0.35 (TRExp for this) The `this` command means the actual class. Therefore the type of the term is the argument *act_cl*.

$\mathsf{TRExp}(\ \overline{\sigma}, V, \mathsf{this}\ ) = \{\ (\overline{\sigma}, act\_cl, V)\ \}$

[1] $A_\theta \backslash (f^{(m)})$ is an abreviation which stands for the set of type assumptions A_θ without the type assumptions for the method f with m arguments.

Algorithm C.0.36 (TRExp for super) The `super` command means the direct super class of the actual class.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{super}\,) =$
let
$(act\_cl, \tau') \in \leq$
in
$\{\,(\overline{\sigma}, \tau', V)\,\}$
end

Algorithm C.0.37 (TRExp for local variables respectively method arguments)
The local variable are either arguments of the method or local variables of the actual block. The type of a local variable is given as the type assumption of the variable in the set V. Therefore the substitution $\overline{\sigma}$ is not extended.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{LocalOrFiledVar}(\,x\,)\,) =$
let
$V = V_{(x,0)} \cup \{\,x : \theta\,\}$
in
$\{\,(\overline{\sigma}, \theta, V)\,\}$
end

Algorithm C.0.38 (TRExp for instance variables) The type reconstruction for instance variables is divided two functions. In the first, the possibly different types of the receiver re are determined.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{InstVar}(\,re, v\,)\,) =$
let
$\{\,result_1, , \ldots, result_l\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, re\,)$
in
$\bigcup_{1 \leqslant \lhd} \mathsf{TRInstVar}(\,result_i, v\,)$
end

For all results of this, the function TRInstVar is called, where the type of the respective instance variable v is determined.

$\mathsf{TRInstVar}(\,(\sigma, \nu, V), v\,) =$
let-foreach $C\texttt{<}a_1, \ldots, a_n\texttt{>}$ with $((v^{(0)} : \theta_C) \in A_{\forall a_1 \ldots . \forall a_n . C\texttt{<}a_1, \ldots, a_n\texttt{>}})$ **or**
$(C\texttt{<}a_1, \ldots, a_n\texttt{>} = act\_cl$ and $(\mathsf{this}.v^{(0)} : \theta_C) \in V))$:
$\theta_0 = \mathsf{fresh}(\,C\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$
$\sigma_C = \mathbf{TUnify}_{\leq^*}(\,\nu, \theta_0\,)$
in
$\bigcup_{C} \{\,(\sigma_C \circ \sigma, \sigma_C(\,\theta_C\,), \mathbf{sub}(\,\sigma_C, V\,))\,\}$
end

The next algorithm determines the type of an array access.

Algorithm C.0.39 (TRExp for the array access) In this algorithm first the types of the index are determined. Only the if the result type is `int`, the result triples are considered further on. For these triples the types of the array are determined. If the type of the array is an array type then a result type of the whole expression is given.

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{ArrayAcc}(\,e, index\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, index\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n \text{ with } ty_i = \texttt{int}\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \theta, V_{i,j}) \mid ty_{i,j} = \theta\,\texttt{[]}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.40 (TRExp for Literals) For the literals of the types `int`, `boolean`, and char the corresponding types are the result. For the literal Null any type is possible. Therefore a new type variable is generated as its type.

$$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{IntLiteral}(\,n\,)\,) = \{\,(\overline{\sigma}, \texttt{int}, V)\,\}$$

$$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{BoolLiteral}(\,b\,)\,) = \{\,(\overline{\sigma}, \texttt{boolean}, V)\,\}$$

$$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{CharLiteral}(\,c\,)\,) = \{\,(\overline{\sigma}, \texttt{char}, V)\,\}$$

$$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Null}\,) = \{\,(\overline{\sigma}, \mathsf{fresh}(\,a\,), V)\,\}$$

Algorithm C.0.41 (TRExp for UnaryMinus) The argument of the UnaryMinus must be `int`. Therefore only the result triples of this type are considered.

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{UnaryMinus}(\,e\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e\,) \\
\quad \textbf{in} \\
\qquad \{\,(\sigma_i, \texttt{int}, V_i) \mid ty_i = \texttt{int}\,\} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.42 (TRExp for Not) The argument of the Not must be `boolean`. Therefore only the result triples of this type are considered.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Not}(\,e\,)\,) =$
let
$\quad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e\,)$
in
$\quad \{\,(\sigma_i, \texttt{boolean}, V_i) \mid ty_i = \texttt{boolean}\,\}$
end

Algorithm C.0.43 (TRExp for Cast) The cast–expression transforms the type of an expression to the given type. Therefore the result type is the given type.

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Cast}(\,\theta, e\,)\,) =$
let
$\quad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,)$
in
$\quad \{\,(\sigma_1, \theta, V_1), \ldots, (\sigma_n, \theta, V_n)\,\}$
end

The following algorithms for the addition (C.0.44), the substraction (C.0.45), the multiplication (C.0.46), the division (C.0.47), and the modolo function (C.0.48) are of the same structure. The two arguments must have the type `int`. The result type is then also `int`. All other types are not considered.

Algorithm C.0.44 (TRExp for Add)

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Add}(\,e_1, e_2\,)\,) =$
let
$\quad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,)$
in
$\quad$ **let-foreach** $1 \leqslant i \leqslant n$ with $ty_i = \texttt{int}$:
$\quad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,)$
$\quad$ **in**
$\quad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \texttt{int}, V_{i,j}) \mid ty_{i,j} = \texttt{int}\,\}$
$\quad$ **end**
end

Algorithm C.0.45 (TRExp for Minus)

$\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Minus}(\,e_1, e_2\,)\,) =$
let
$\quad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,)$
in
$\quad$ **let-foreach** $1 \leqslant i \leqslant n$ with $ty_i = \texttt{int}$:
$\quad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,)$
$\quad$ **in**

$$\bigcup_{ty_i = \texttt{int}} \{\, (\sigma_{i,j}, \texttt{int}, V_{i,j}) \mid ty_{i,j} = \texttt{int} \,\}$$

end

end

Algorithm C.0.46 (TRExp for Mul)

$\mathsf{TRExp}(\, \overline{\sigma}, V, \mathsf{Mul}(\, e_1, e_2 \,)\,) =$

let

$\{\, (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \,\} = \mathsf{TRExp}(\, \overline{\sigma}, V, e_1 \,)$

in

let-foreach $1 \leqslant i \leqslant n$ with $ty_i = \texttt{int}$:

$\{\, (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \,\} = \mathsf{TRExp}(\, \sigma_i, V_i, e_2 \,)$

in

$$\bigcup_{ty_i = \texttt{int}} \{\, (\sigma_{i,j}, \texttt{int}, V_{i,j}) \mid ty_{i,j} = \texttt{int} \,\}$$

end

end

Algorithm C.0.47 (TRExp for Div)

$\mathsf{TRExp}(\, \overline{\sigma}, V, \mathsf{Div}(\, e_1, e_2 \,)\,) =$

let

$\{\, (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \,\} = \mathsf{TRExp}(\, \overline{\sigma}, V, e_1 \,)$

in

let-foreach $1 \leqslant i \leqslant n$ with $ty_i = \texttt{int}$:

$\{\, (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \,\} = \mathsf{TRExp}(\, \sigma_i, V_i, e_2 \,)$

in

$$\bigcup_{ty_i = \texttt{int}} \{\, (\sigma_{i,j}, \texttt{int}, V_{i,j}) \mid ty_{i,j} = \texttt{int} \,\}$$

end

end

Algorithm C.0.48 (TRExp for Mod)

$\mathsf{TRExp}(\, \overline{\sigma}, V, \mathsf{Mod}(\, e_1, e_2 \,)\,) =$

let

$\{\, (\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n) \,\} = \mathsf{TRExp}(\, \overline{\sigma}, V, e_1 \,)$

in

let-foreach $1 \leqslant i \leqslant n$ with $ty_i = \texttt{int}$:

$\{\, (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \,\} = \mathsf{TRExp}(\, \sigma_i, V_i, e_2 \,)$

in

$$\bigcup_{ty_i = \texttt{int}} \{\, (\sigma_{i,j}, \texttt{int}, V_{i,j}) \mid ty_{i,j} = \texttt{int} \,\}$$

end

end

The following algorithms for Less (C.0.49), LessEq (C.0.50), Greater (C.0.51), and GreaterEq (C.0.52) are very similar to the above algorithms. Only the result type is not int, but `boolean`.

Algorithm C.0.49 (TRExp for Less)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Less}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n \text{ with } ty_i = \texttt{int}\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \texttt{boolean}, V_{i,j}) \mid ty_{i,j} = \texttt{int}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.50 (TRExp for LessEq)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{LessEq}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n \text{ with } ty_i = \texttt{int}\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \texttt{boolean}, V_{i,j}) \mid ty_{i,j} = \texttt{int}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.51 (TRExp for Greater)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Greater}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n \text{ with } ty_i = \texttt{int}\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \texttt{boolean}, V_{i,j}) \mid ty_{i,j} = \texttt{int}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.52 (TRExp for GreaterEq)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{GreaterEq}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n \text{ with } ty_i = \texttt{int}\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{ty_i = \texttt{int}} \{\,(\sigma_{i,j}, \texttt{boolean}, V_{i,j}) \mid ty_{i,j} = \texttt{int}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

The next two algorithms Equal (C.0.53) and NotEqual (C.0.54) allows as argument type each type. The only condition is, that both arguments have the same type. Therefore the result types of both arguments are unified. If they are successfully unified, the result type is boolean.

Algorithm C.0.53 (TRExp for Equal)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{Equal}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m_i\text{:} \\
\qquad\quad \mathit{unify}_{i,j} = \mathbf{TUnify}_{\leq^*}(\,ty_i, ty_{i,j}\,) \cup \mathbf{TUnify}_{\leq^*}(\,ty_{i,j}, ty_i\,) \\
\qquad \textbf{in} \\
\qquad\quad \bigcup_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m_i}} \{\,(\sigma, \texttt{boolean}, \mathbf{sub}(\,\sigma, V_{i,j}\,)) \mid \sigma \in \mathit{unify}_{i,j}\,\} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Algorithm C.0.54 (TRExp for NEq)

$$
\begin{array}{l}
\mathsf{TRExp}(\,\overline{\sigma}, V, \mathsf{NEq}(\,e_1, e_2\,)\,) = \\
\quad \textbf{let} \\
\qquad \{\,(\sigma_1, ty_1, V_1), \ldots, (\sigma_n, ty_n, V_n)\,\} = \mathsf{TRExp}(\,\overline{\sigma}, V, e_1\,) \\
\quad \textbf{in} \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n\text{:} \\
\qquad\quad \{\,(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \ldots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i})\,\} = \mathsf{TRExp}(\,\sigma_i, V_i, e_2\,) \\
\qquad \textbf{let-foreach } 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m_i\text{:} \\
\qquad\quad \mathit{unify}_{i,j} = \mathbf{TUnify}_{\leq^*}(\,ty_i, ty_{i,j}\,) \cup \mathbf{TUnify}_{\leq^*}(\,ty_{i,j}, ty_i\,)
\end{array}
$$

in
$$\bigcup_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m_i}} \{\, (\sigma, \texttt{boolean}, \mathbf{sub}(\,\sigma, V_{i,j}\,)) \mid \sigma \in \mathit{unify}_{i,j} \,\}$$
end
end

Algorithm C.0.55 (**TUnify**) The function **TUnify** is the type unification algorithm, which is presented in section **??**. Here we describe the adaption of the call of **TUnify** in the type reconstruction algorithm to the presentation in algorithm **??**.
The call

$$\mathbf{TUnify}_{\leq^*}\,(\,(ty_{1,1} \ldots ty_{1,n}), (ty_{2,1} \ldots ty_{2,n})\,)$$

generates the input for algorithm **??**:

$$\{\, (ty_{1,1} \lessdot ty_{2,1}), \ldots, (ty_{1,n} \lessdot ty_{2,n}) \,\}.$$

The set of type variables TV is given as the type variables which are generate by fresh and in NewTVar.
The result is the set of type unifiers which are determined by algorithm **??**.

Algorithm C.0.56 (**sub**) The function $\mathbf{sub}(\,\sigma, V\,)$ substitutes the type variables of the types of the variables and the methods by σ in the set of type assumptions V.

$\mathbf{sub}(\,\sigma, V\,) =$
let
$V = \{\, me_1 : ty_{1,1} \times \ldots \times ty_{1,m_1} \to rty_1, \ldots, me_n : ty_{n,1} \times \ldots \times ty_{n,m_n} \to rty_n \,\}$
$\quad \cup \{\, v_1 \mapsto \nu_1, \ldots, v_p \mapsto \nu_p \,\}$
in
$\quad \{\; me_1 : \sigma(\,ty_{1,1}\,) \times \ldots \times \sigma(\,ty_{1,m_1}\,) \to \sigma(\,rty_1\,), \ldots,$
$\quad\; me_n : \sigma(\,ty_{n,1}\,) \times \ldots \times \sigma(\,ty_{n,m_n}\,) \to \sigma(\,rty_n\,) \,\}$
$\cup \{\; v_1 \mapsto \sigma(\,\nu_1\,), \ldots, v_p \mapsto \sigma(\,\nu_p\,) \,\}$
end

Algorithm C.0.57 (**bind**) The function **bind** takes the free variables of a type assumption of a method, and substitute them by the parameters of the respective class.

$\mathbf{bind}(\,Para, V\,) =$
let
$V = \{\, me : ty_1 \times \ldots \times ty_m \to rty \,\} \cup V'$
$Var = \mathsf{TVar}(\,\{\, ty_1, \ldots, ty_m, rty \,\}\,)$
$subst = \{\, \sigma \mid \sigma : Var \to Para \text{ is a function} \,\}$
in
$\{\, me : \bigwedge_{\sigma \in subst} \sigma(\,ty_1\,) \times \ldots \times \sigma(\,ty_m\,) \to \sigma(\,rty\,) \,\} \cup V'$
end

Abkürzungsverzeichnis

3GL	Third Generation Language
API	Application Programming Interface
CVS	Concurrent Versions System
IDE	Integrated Development Environment
JDK	Java Development Kit
JIT	Just-In-Time-Kompilierung
JVM	Java Virtual Machine
LaTeX	Leslie-Lamport-TeX
TRA	Typrekonstruktionsalgorithmus

Literaturverzeichnis

[AVA99] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilerbau Teil 1*. Oldenbourg Verlag, 1999.

[Bra04] Gilad Bracha. *Generics in the Java Programming Language*. SUN Tutorials, 2004.

[Eck00] Bruce Eckel. *Thinking in C++*. Prentice Hall, 2000.

[Ehl04] Carsten Ehlers. *Statische Typüberprüfung am Beispiel von ML und Haskell*. http://www.fh-wedel.de/ si/seminare/, 2004.

[Fla00] David Flanagan. *Java in a Nutshell*. O'Reilly, 2000.

[Haa04] Markus Haas. *Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen*. BA Horb, 2004.

[oF95] Gang of Four. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[Plü99] Martin Plümicke. *The Polymorphic Extension of OBJ-3*. Universität Tübingen, 1999.

[Plü04a] Dr. Martin Plümicke. *Type Inference in Generic Java*, 2004.

[Plü04b] Dr. Martin Plümicke. *Vorlesungsskript: Infrastruktur und Betriebssysteme III*, 2004.

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

[uKK04a] Angelika Langer und Klaus Kreft. *Java Generics und Type Erasure - Parametrisierte Typen und Methoden*. Java Magazin, April 2004.

[uKK04b] Angelika Langer und Klaus Kreft. *Java Generics und Type Erasure - Umwälzungen im Java-Typsystem*. Java Magazin, Oktober 2004.

[uRH98] Dr. Bernhard Bauer und Riitta Höllerer. *Übersetzung objektorientierter Programmiersprachen*. Springer-Verlag, 1998.

[Wik05] Wikipedia. *Die freie Enzyklopädie*. htpp://www.wikipedia.de, 2005.