Java-TX Compiler in Java-TX

Studienarbeit (Modul T3_3101)

Studiengang Informatik

Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

von

Julian Schmidt

Juni 2024

Matrikelnummer, Kurs 3279342, HOR-TINF2021

Betreuer/Gutachter Prof. Dr. habil. Martin Plümicke

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit (Modul T3_3101) mit dem Thema: *Java-TX Compiler in Java-TX* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Horb am Neckar, 02. Juni 2024

_____

Julian Schmidt

Abstract

Seit rund 15 Jahren wird an der Dualen Hochschule Baden Württemberg Campus Horb an einer Java-Erweiterung namens Java-TX (Type eXtended) auf Basis von Java 8 geforscht. Java-TX zielt darauf ab, die Java-Programmiersprache durch funktionale Programmierkonzepte wie globale Typinferenz und echte Funktionstypen für Lambda-Ausdrücke zu erweitern. Im Bereich des Compilerbaus gilt die Selbstübersetzung eines Compilers als wichtiges Qualitätsmerkmal. Aus diesem Grund wird im Rahmen dieser Arbeit der in Java verfassten Java-TX-Compiler so weit es möglich ist, in Java-TX übersetzt. Dabei werden der Funktionsumfang und die Praxistauglichkeit des aktuellen Java-TX-Zustands untersucht und bestehende Probleme sowie fehlende Funktionen aufgezeigt. Zusätzlich werden durch die Überarbeitung des Quellcodes die Vorteile von Java-TX im Vergleich zu Java demonstriert.

Inhaltsverzeichnis

Abkürzungsverzeichnis V

Abbildungsverzeichnis VI

Tabellenverzeichnis VII

Listings VIII

1 Grundlagen 1
- 1.1 Typinferenz in Java . . . 1
  - 1.1.1 Typinferenz bei generischen Methoden . . . 1
  - 1.1.2 Der Diamond Operator . . . 2
  - 1.1.3 Typinferenz bei Lambda Ausdrücken . . . 3
  - 1.1.4 Der Typplatzhalter var . . . 3
- 1.2 Typinferenz in Java-TX . . . 4
- 1.3 Anonyme Funktionen in Java . . . 6
  - 1.3.1 Funktionale Interfaces und Lambda Ausdrücke . . . 7
  - 1.3.2 Wildcards . . . 10
- 1.4 Echte Funktionstypen in Java-TX . . . 11
- 1.5 GNU Make . . . 12
- 1.6 Selbstkompilierende Compiler . . . 14

2 Aufbau der Umgebung 16
- 2.1 Voraussetzungen . . . 16
- 2.2 Kompilierung mit Make . . . 18
  - 2.2.1 Performanceprobleme . . . 20
- 2.3 Kompilierung mit Bash . . . 21
- 2.4 Tests . . . 23

3 Aufgetretene Probleme 26
- 3.1 Neue Funktionen . . . 27
  - 3.1.1 ForEach Schleife . . . 27
  - 3.1.2 Weitere neue Funktionen . . . 28
- 3.2 Bugs . . . 28
  - 3.2.1 JVM Classpath wird von „Java-TX Compiler“ beachtet . . . 29
  - 3.2.2 Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces . . . 31
  - 3.2.3 Überschreiben von Methoden mit primitiven Datentypen . . . 34
  - 3.2.4 Korrekter Methodenaufruf für überladene Methoden mit Subtypen als Parameter . . . 37

3.2.5 Weitere Bugs und fehlende Features 39

4 Vorteile in der Praxis **40**

5 Fazit und Ausblick **44**
5.1 Fazit 44
5.2 Ausblick 45

Literatur **46**

Anhang **51**
.1 Sourcecode des Bash Skripts zur Kompilierung 51

Abkürzungsverzeichnis

Java-TX	Java-Type eXtended
GNU	GNU's Not Unix
JVM	Java Virtual Machine
JDK	Java Development Kit
WSL	Windows Subsystem for Linux
IDE	Integrated Development Environment
GCC	GNU Compiler Collection
Bash	Bourne Again Shell

Abbildungsverzeichnis

1.1 Selbstkompilierender Compiler in Java-TX 15

2.1 Zirkuläre Abhängigkeiten zwischen Java und Java-Type eXtended (Java-TX) Dateien 16
2.2 Zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien behoben . . 17
2.3 Dateistruktur des Projekts 22

3.1 Gefundene Probleme und neue Funktionen 26
3.2 Die Classloader Hierarchie des „Java-TX Compiler“ (vgl. [18]) 30
3.3 Die Classloader Hierarchie des „Java-TX Compiler“ ohne den Application-ClassLoader 31
3.4 Primitive Datentypen in Java-TX 35

4.1 Vererbungshierarchie von java.util.List ab Java 21 [13] 42

5.1 Verhältnis der Java und Java-TX Dateien im „Java-TX Compiler in Java-TX“ 44

Tabellenverzeichnis

1.1 Gängige funktionale Interfaces in der Java Standardbibliothek 9

2.1 Kompilierzeiten des „Java-TX Compiler“ mit javac 21

Listings

1.1 List.of() mit explizitem Typ . . . 2
1.2 List.of() mit implizitem Typ . . . 2
1.3 Verbesserungen für Typinferenz in Java 8 (Beispiel aus [21][Target Types]) 2
1.4 Java 7 Diamond Operator . . . 2
1.5 Java Generics ohne Diamond Operator . . . 3
1.6 Java 8 Lambda-Ausdrücke . . . 3
1.7 Lambda-Ausdruck mit explizitem Typ . . . 3
1.8 Java 10 var Platzhalter . . . 4
1.9 Ungültige Verwendung des `var` Schlüsselworts . . . 4
1.10 Funktion add in Haskell . . . 4
1.11 Untypisierte Methode add . . . 5
1.12 Von Typinferenz errechnete Typen für Listing 1.11 . . . 5
1.13 Methodenüberladungen durch Typinferenz . . . 6
1.14 Resultat der Typinferenz für Listing 1.13 . . . 6
1.15 Erstellung eines Threads mit einer anonymen Klasse . . . 7
1.16 Erstellung eines Threads mit einem Lambda Ausdruck . . . 8
1.17 Lambda Ausdruck mit var . . . 8
1.18 Varianz in Java . . . 11
1.19 Subtypisierung von Funktionstypen in Java-TX . . . 12
1.20 Lambda Ausdruck ohne Typkontext . . . 12
1.21 Aufbau einer Makefile-Regel aus [30][S.3] . . . 12
1.22 Beispiel eines Makefiles aus [30][S.4] . . . 13

2.1 Makefile für die Kompilierung des „Java-TX Compiler in Java-TX“ . . . 18
2.2 Skript zum Kompilieren und Ausführen der Tests . . . 24

3.1 For-Each Schleife in Java . . . 27
3.2 For-Each Schleife in Java-TX . . . 27
3.3 Verwenden von Klassen im JVM Classpath . . . 30
3.4 Verwendung der Stream API in Java . . . 32
3.5 Aktuell nicht lauffähiger Java-TX Code I . . . 33
3.6 Aktuell nicht lauffähiger Java-TX Code II . . . 34
3.7 Überschreiben von Methoden mit primitiven Datentypen in Java-TX . . . 35
3.8 Ergebnis der Typinferenz für die Methode `hashCode` in Java-TX . . . 35
3.9 Kovariante Methodenüberladung in Java . . . 36

3.10 Dekompilierter Bytecode der Klasse A 37
3.11 Überladene Methoden in Java 38

4.1 Beispielklasse aus dem „Java-TX Compiler“ 40
4.2 Listing 4.1 ohne Typinformationen 41
4.3 Inferierte Typen für Listing 4.2 42

1 Grundlagen

Java-TX basiert auf Java 8 und wurde um zwei wesentliche Funktionen erweitert, die man aus funktionalen Programmiersprachen wie Haskell kennt: Globale Typinferenz und echte Funktionstypen. Im folgenden Kapitel werden diese beiden Konzepte genauer erläutert. Zunächst soll aber ein Einblick gegeben werden, was bezüglich dieser Funktionen in Java möglich ist.

1.1 Typinferenz in Java

Java unterstützt verschiedene Arten lokaler Typinferenz. Das bedeutet, dass der Programmierer an bestimmten Stellen des Codes den Datentyp einer Variable nicht explizit angeben muss, sondern der Compiler diesen anhand des Kontext ableiten kann. In diesem Abschnitt werden einige dieser Mechanismen vorgestellt.

1.1.1 Typinferenz bei generischen Methoden

Mit der Einführung von generischen Methoden in Java 5 [3], wurde auch eine einfache Form der Typinferenz für die Parameter dieser Methoden eingeführt [10][S.451 ff]. Konkret ist es möglich, den Compiler einen generischen Typ einer Methode anhand der Parametertypen inferieren zu lassen. Die Methode `of` in `java.util.List` dient als Beispiel zur Veranschaulichung für diese Art von Typinferenz. Sie erzeugt eine Liste des generischen Typs `T` mit dem übergebenen Parameter als Element. Im folgenden ist die Signatur dieser Methode für einen einzelnen Parameter[1] dargestellt.

```
public interface List<E> extends SequencedCollection<E> {
    ...
    static <E> List<E> of(E e1);
    ...
}
```

[1] Es gibt mehrere Versionen der Methode `of` in `java.util.List` mit einer unterschiedlichen Anzahl an Parametern

Ohne Typinferenz müsste man der Methode beim Aufruf explizit den generischen Typen angeben, wie in Listing 1.1 gezeigt ist.

```
1  List<Integer> intList = List.<Integer>of(1);
```

Listing 1.1: List.of() mit explizitem Typ

Die Typinferenz erlaubt es allerdings diese explizite Typangabe wegzulassen, da der generische Typ `T` implizit aus den Typen der Parameterliste resultiert. Dieser kompaktere Aufruf ist in Listing 1.2 zu sehen.

```
1  List<Integer> intList = List.of(1);
```

Listing 1.2: List.of() mit implizitem Typ

Mit der Veröffentlichung von Java 8 wurde diese Form der Typinferenz weiter verbessert und erweitert [4], so dass nun auch komplexere Typinferenz, wie in Listing 1.3 gezeigt wird, möglich ist. Vor Java 8 war dies nicht möglich [4, 21].

```
1   public class A{
2       public static void main(String[] args){
3           // In Java 7: error: incompatible types: List<Object>
                cannot be converted to List<String>
4           // In Java 8: korrekt
5           processStringList(Collections.emptyList());
6       }
7
8       void processStringList(List<String> list){
9           //...
10      }
11  }
```

Listing 1.3: Verbesserungen für Typinferenz in Java 8 (Beispiel aus [21][Target Types])

1.1.2 Der Diamond Operator

Java unterstützt seit Version 7 den sogenannten Diamond Operator <> [5]. Dieser ermöglicht es, den Typ beim Instanziieren einer generischen Klasse nicht explizit anzugeben, sondern vom Compiler errechnen zu lassen [11][S.482 ff]. Ein Beispiel dafür ist in Listing 1.4 zu sehen. Der Typ der Liste auf der rechten Seite der Zuweisung muss nicht explizit angegeben werden, sondern kann vom Compiler anhand des Kontextes abgeleitet werden.

```
1   List<String> list = new ArrayList<>();
```

Listing 1.4: Java 7 Diamond Operator

Vor Java 7 musste dieser Typ explizit angegeben werden, wie in Listing 1.5 zu sehen ist.

```
1     List<String> list = new ArrayList<String>();
```

Listing 1.5: Java Generics ohne Diamond Operator

Erwähnt sei hierbei auch, dass `List<String> = new ArrayList();` nicht das gleiche ist wie `List<String> = new ArrayList<>();` ist. Im ersten Fall wird die Liste als Raw Type erstellt [21][Type Inference and Instantiation of Generic Classes]. Aus diesem Grund ist die Angabe des Diamond Operators in diesem Fall wichtig, um den generischen Typen zu erhalten.

1.1.3 Typinferenz bei Lambda Ausdrücken

Auch in Lambda-Ausdrücken, welche in Version 8 hinzugefügt wurden, erlaubt Java lokale Typinferenz [6]. Der Typ der Parameter einer Lambda-Funktion kann vom Compiler anhand des Kontextes abgeleitet werden [11][S.603 ff], wie in Listing 1.6 zu sehen ist. Die Methode `forEach` in Zeile 2 wendet dabei einen gegebenen Lambda Ausdruck auf jeden Eintrag der Liste an. In diesem Fall würde sie jedes Element der Liste ausgeben. Auf Lambda-Ausdrücke wird in Abschnitt 1.3 ausführlich eingegangen. Wie man sehen kann, muss der Typ des Parameters `s` in Zeile 2 vom Programmierer nicht explizit angegeben werden, da der Compiler weiß, dass es sich bei der Liste `list` um eine Liste von Strings handelt. Ein Element der Liste muss somit ein String sein.

```
1 List<String> list = new ArrayList<>();
2 list.forEach(s -> System.out.println(s));
```

Listing 1.6: Java 8 Lambda-Ausdrücke

Es steht dem Entwickler jedoch frei den Typ des Parameters explizit anzugeben, wie in Listing 1.7 gezeigt ist.

```
1 List<String> list = new ArrayList<>();
2 list.forEach((String s) -> System.out.println(s));
```

Listing 1.7: Lambda-Ausdruck mit explizitem Typ

1.1.4 Der Typplatzhalter var

Ein weiteres Beispiel für lokale Typinferenz ist der Platzhalter `var`, welcher mit Java 10 eingeführt wurde [8]. Er ermöglicht den Typ einer lokalen Variablen nicht explizit anzugeben zu müssen, wenn dieser unmittelbar aus dem Kontext hervorgeht [8]. Ein Beispiel dafür ist in Listing 1.8 zu sehen.

```
var list = new ArrayList<String>();
```

Listing 1.8: Java 10 var Platzhalter

Da der Compiler für den Typcheck den Typ der rechten Seite der Zuweisung ohnehin berechnen muss, ist es ein Leichtes den `var` Platzhalter zur Kompilierzeit durch den tatsächlichen Typ zu ersetzen. Im Beispiel von oben wäre dies der Typ `ArrayList<String>`. Die Möglichkeiten sind jedoch schnell erschöpft. So ist es nicht möglich, den Typ einer Methode oder eines Feldes mit `var` zu deklarieren. Außerdem muss bei der Verwendung die Initialisierung der Variable mit der Deklaration erfolgen [8]. Eine Zuweisung zum Literal „null“ ist demnach ebenfalls ungültig. Diese Einschränkungen sind in Listing 1.9 zu sehen.

```
//Ungueltiger Code
var x;
x = 10;
//Ungueltiger Code
var y = null;
```

Listing 1.9: Ungültige Verwendung des `var` Schlüsselworts

1.2 Typinferenz in Java-TX

Trotz verschiedener Arten der lokalen Typinferenz, die in den letzten Jahren zu Java hinzugefügt wurden, müssen bis heute die Typen in Methodensignaturen und Feldern explizit vom Entwickler angegeben werden. In funktionalen Programmiersprachen wie Haskell existiert hingegen eine globale Typinferenz. Diese erlaubt es auch die Typen der Parametern und des Rückgabewerts einer Methode vom Compiler ableiten zu lassen [34][S.323 ff]. In Listing 1.10 ist die Funktion `add`, welche zwei numerische Werte addiert, in Haskell gegeben. Wie man sehen kann, beinhaltet der Quellcode keinerlei Typinformationen. Dennoch ist Haskell eine statisch typisierte Sprache [15][S.3]. Das bedeutet, dass sämtliche Typinformationen bereits zur Kompilierzeit überprüft werden [22][S.2]. Der Haskell Compiler inferiert die Typen also zur Kompilierzeit. Im Fall der Funktion `add`, wird der Typ `add :: Num a => (a, a)-> a` errechnet. Dieser Typ bedeutet, dass die Funktion `add` zwei Werte vom Typ `a` entgegennimmt und einen Wert vom selben Typ zurückgibt. `a` muss dabei von der Typklasse `Num` sein, was in Haskell die numerischen Typen einschließt [15][S.76, 81]. Die Funktion `add` kann also sowohl mit ganzen Zahlen, als auch mit Gleitkommazahlen aufgerufen werden.

```
add (x, y) = x+y
```

Listing 1.10: Funktion add in Haskell

Eines der primären Ziele von Java-TX ist es daher, die globale Typinferenz in Java zu ermöglichen, um auch Typen von Feldern und Methodensignaturen vom Compiler inferieren lassen zu können.

Ein Beispiel für die Funktion `add` in Java-TX ist in Listing 1.11 zu sehen.

```
1 import java.lang.Integer;
2 class Add{
3     public add(a, b) {
4         return a + b;
5     }
6 }
```

Listing 1.11: Untypisierte Methode add

In diesem Beispiel werden ähnlich wie in Listing 1.10 sowohl die Parametertypen, als auch der Rückgabetyp nicht angegeben. Diese Typen werden zur Kompilierzeit anhand des Kontextes abgeleitet. Die resultierenden Typen der Methode sind in Listing 1.12 gegeben. Der Compiler weiß, dass der Operator `+` nur für numerische Datentypen und Zeichenketten definiert ist. Da der Algorithmus für die globalen Typinferenz allerdings sehr rechenintensiv ist, werden nur explizit importierte Typen berücksichtigt. In diesem Beispiel wird nur `java.lang.Integer` importiert, daher spielen andere numerische Typen und Zeichenketten keine Rolle. Neben den Parametern berechnet der Compiler auch einen möglichen Rückgabetyp. In diesem Fall hat dieser auch den Datentyp `java.lang.Integer`, da die Addition von zwei Integern wieder einen Integer ergibt.

```
1 import java.lang.Integer;
2 class Add{
3     public Integer add(Integer a, Integer b) {
4         return a + b;
5     }
6 }
```

Listing 1.12: Von Typinferenz errechnete Typen für Listing 1.11

Ähnlich wie C++ Templates erlaubt dieses Vorgehen es Methoden für verschiedene Datentypen mit gleicher Implementierung nur einmal zu definieren. Der Unterschied besteht darin, dass Java-TX den Code für alle Typen, die importiert wurden und die verwendeten Funktionalitäten unterstützen, generiert. C++ generiert den Code nur für die Typen, mit denen das Template explizit aufgerufen wird [35][Abschnitt 2.1.2]. Dies ermöglicht es, den Code zu vereinfachen und die Wartbarkeit zu erhöhen. Ein Beispiel dazu ist in Listing 1.13 zu sehen.

```
import java.lang.Integer;
import java.lang.String;
import java.lang.Double;

public class Add{
  public add(a, b) {
    return a + b;
  }
}
```

Listing 1.13: Methodenüberladungen durch Typinferenz

In diesem Fall erzeugt der Compiler die Methode `add` jeweils mit der Signatur für die Datentypen `java.lang.Integer`, `java.lang.String` und `java.lang.Double`, wie in Listing 1.14 gezeigt ist. Denkbar wären mit diesen Imports auch weitere valide Methodensignaturen wie `Double add(Integer a, Double b)` oder `Double add(Double a, Integer b)`, welche allerdings aktuell nicht untersützt werden.

```
import java.lang.Integer;
import java.lang.String;
import java.lang.Double;
public class Add{
  public Integer add(Integer a, Integer b) {
    return a + b;
  }

  public Double add(Double a, Double b) {
    return a + b;
  }

  public String add(String a, String b) {
    return a + b;
  }
}
```

Listing 1.14: Resultat der Typinferenz für Listing 1.13

1.3 Anonyme Funktionen in Java

Durch die Arbeiten an „Project Lambda“ [20] wurden anonyme Funktionen, sogenannte Lambda Ausdrücke, in Java 8 implementiert [9]. Lambda Ausdrücke ermöglichen in Java

eine kompaktere Schreibweise, um Interfaces zu implementieren, welche nur eine einzige abstrakte Methode enthalten. Diese Interfaces werden als funktionale Interfaces bezeichnet [11][S.321 ff].

1.3.1 Funktionale Interfaces und Lambda Ausdrücke

Ein Beispiel für ein funktionales Interface ist das Interface `java.lang.Runnable`, welches nur eine abstrakte Methode mit dem Namen `run` enthält. `run` hat keine Parameter und keinen Rückgabewert. Eine Implemetierung dieses Interfaces kann z.B. verwendet werden, um das Verhalten eines Threads zu definieren. Vor der Einführung von Lambda Ausdrücken, war die einfachste Möglichkeit dies zu tun die Verwendung von anonymen Klassen [11][S.491 ff]. Dies ist in Listing 1.15 zu sehen.

```
1  class Main{
2    public void startThread(){
3        Thread t = new Thread(new Runnable() {
4            @Override
5            public void run() {
6                System.out.println("Hello from Thread");
7            }
8        });
9        t.start();
10   }
11 }
```

Listing 1.15: Erstellung eines Threads mit einer anonymen Klasse

Durch diese Verwendung einer anonymen Klasse, kann eine spezielle Implementierung der Methode `run()` erstellt werden, ohne eine neue Klasse zu erstellen, die das Interface implementiert. Allerdings erfordert diese Lösung viel Boilerplate Code, welcher die Lesbarkeit des Codes erschwert. Die eigentliche Logik des Threads ist lediglich durch den Aufruf der `println` Methode in Zeile 6 definiert.

Durch die Einführung von Lambda Ausdrücken kann der benötigte Code für diesen Anwendungsfall deutlich reduziert werden. Ein Lambda Ausdruck kann ein beliebiges funktionales Interface, bzw. dessen abstrakte Methode implementieren. Listing 1.16 zeigt wie der Code in Listing 1.15 durch die Verwendung von Lambda ausdrücken reduziert werden kann.

```
class Main{
  public void startThread(){
    Thread t = new Thread(() -> System.out.println("Hello from
       Thread"));
    t.start();
  }
}
```

Listing 1.16: Erstellung eines Threads mit einem Lambda Ausdruck

Wie man sehen kann benötigt man für die gleiche Funktionalität nun nur noch 6, statt 11 Zeilen Code. Zusätzlich ist der Code auf das Wesentliche reduziert, wodurch die Lesbarkeit verbessert wurde.

Wie bereits erwähnt, haben Lambda Ausdrücke in Java keinen eigenen Typen, sondern werden immer einem funktionalen Interface zugeordnet. Man spricht in diesem Kontext von sogenannten Zieltypen (engl. target types). Der Typ muss immer aus dem Kontext hervorgehen. In Listing 1.16 wird der Lambda Ausdruck dem Interface `java.lang.Runnable` zugeordnet. Der Compiler erkennt, dass der Konstruktor der Klasse Thread das funktionale Interface `java.lang.Runnable` erwartet und kann somit den Lambda Ausdruck diesem Interface zuordnen. Auf Grund dieser Tatsache ist der Code in Listing 1.17 nicht gültig. Der Compiler kann den Typ nicht errechnen, da der Lambda Ausdruck unendlich viele Typen annehmen kann. Dies sind alle funktionalen Interfaces, deren abstrakte Methode keinen Parameter erwarten und `void` zurückgeben.

```
  //Ungueltige Syntax
  var lambda = () -> System.out.println("Hello World!");
```

Listing 1.17: Lambda Ausdruck mit var

Die Java Standardbibliothek stellt dem Entwickler bereits einige funktionale Interfaces zur Verfügung. Einige davon sind in Tabelle 1.1 aufgelistet.

Interface	**Abstrakte Methode**
java.lang.Runnable	void run()
java.util.concurrent.Callable<V>	V call()
java.util.function.Consumer<T>	void accept(T t)
java.util.function.Supplier<T>	T get()
java.util.function.Function<T, R>	R apply(T t)
java.util.function.Predicate<T>	boolean test(T t)

Tabelle 1.1: Gängige funktionale Interfaces in der Java Standardbibliothek

Wie man Tabelle 1.1 entnehmen kann, sind die funktionalen Interfaces der Java Standardbibliothek mit generischen Typparametern definiert. So können die Interfaces für beliebige Referenztypen verwendet werden. Dies bedeutet allerdings auch das die funktionalen Interfaces invariant sind.

Bevor diese Eigenschaft aber näher spezifiziert wird, müssen einige Begriffe geklärt werden, die in diesem Zusammenhang relevant sind. Invarianz, Kovarianz und Kontravarianz sind Begriffe, die in der Typtheorie verwendet werden, um die Beziehung zwischen Typen zu beschreiben. Die folgenden Definitionen sind angelehnt an [12][Abschnitt 1].

Sei C<T> ein generischer Typ mit dem Typparameter T und A, B Typen mit A $\leq^*$[1] B und A $\neq$ B.

Definition 1.3.1 (Invarianz)**.** C<T> ist invariant im Bezug auf den Typparameter T, wenn `C<A>` $\not\leq^*$ `C<B>` und `C<B>` $\not\leq^*$ `C<A>`

Definition 1.3.2 (Kovarianz)**.** C<T> ist kovariant im Bezug auf den Typparameter T, wenn `C<A>` $\leq^*$ `C<B>`.

Definition 1.3.3 (Kontravarianz)**.** C<T> ist kontravariant im Bezug auf den Typparameter T, wenn `C<A>` $\geq^*$ `C<B>`.

Definition 1.3.4 (Varianz)**.** C<T> ist variant im Bezug auf den Typparameter T, wenn er entweder kovariant oder kontravariant ist.

[1] $\leq^*$ steht in diesem Dokument für die Subtyp Beziehung, d.h. A ist ein Subtyp von B

Generell ist zu erwähnen, dass die Eingabewerte eines Funktionstypen kontravariant und der Rückgabewert kovariant ist. Nach [25] gilt also folgende Definition für die Subtypisierung von Funktionstypen.

Definition 1.3.5 (Subtypisierung für Funktionstypen)**.** Gegeben seien die Funktionstypen

$$(\tau_1, \tau_2, ..., \tau_n) \to \tau_0$$
$$(\theta_1, \theta_2, ..., \theta_n) \to \theta_0$$

Es gilt:

$$(\tau_1, \tau_2, ..., \tau_n) \to \tau_0 \leq^* (\theta_1, \theta_2, ..., \theta_n) \to \theta_0$$

$$\forall i \in \{1, \ldots, n\} : \theta_i \leq^* \tau_i$$
$$\text{und } \tau_0 \leq^* \theta_0$$

Generische Typen sind in Java invariant [17][S.15 ff]. Daraus folgt unter anderem auch, dass Subtypisierung nicht wie erwartet möglich ist, z.B. ist

`List<Integer>` $\not\leq^*$ `List<Number>`

Diese Einschränkung besteht auch für Funktionstypen. Demnach ist

`Function<Integer, Integer>` $\not\leq^*$ `Function<Integer, Number>`

obwohl dies nach 1.3.5 eine valide Subtypisierung wäre.

1.3.2 Wildcards

Um das Subtyp Problem für generische Datentypen zu lösen, wird in Java mit den sogenannten Wildcards use site Varianz ermöglicht. Wildcards bieten die Möglichkeit generische Typparameter kovariant oder kontravariant zu verwenden. Kovarianz wird dabei mit `? extends` und Kontravarianz mit `? super` erreicht [17][S.17 ff]. So ist durch die Verwendung von Wildcards folgende Subtypisierung möglich:

`Function<Integer, Integer>` $\leq^*$ `Function<? super Integer, ? extends Number>`

Als Java Quellcode ist dieses Beispiel in Listing 1.18 dargestellt.

```
Function<Integer, Integer> func1 = x -> x + 1;
Function<? super Integer, ? extends Number> func2 = func1;
```

Listing 1.18: Varianz in Java

Wie man sehen kann, wird die Funktion `func1` der Funktion `func2` zugewiesen. Daraus folgt, dass `func2` ein Subtyp von `func1` ist. Dabei ist der Parameter der Funktion `func2` ein Supertyp des Parameters der Funktion `func1` (Kontravarianz) und der Rückgabewert der Funktion `func2` ein Subtyp von dem Rückgabewert der Funktion `func1` (Kovarianz). Dies entspricht der Definition für die Subtypisierung für Funktionstypen aus 1.3.5. Use site Varianz führt zwar zu einer erhöhten Flexibilität, da der generische Typ an verschiedenen Stellen im Code unterschiedlich variiert werden kann, aber es erschwert auch die Lesbarkeit des Codes. Außerdem muss sich der Entwickler um die korrekte Verwendung der Varianz kümmern, was leicht zu Typfehlern führen kann. Einfacher wäre es, wenn die Varianz direkt bei der Deklaration der generischen Typen festgelegt werden könnte. Dies wird als declaration site Varianz bezeichnet und wird von vielen Sprachen wie Scala oder C# unterstützt [23, 36]. Declaration site Varianz ist allerdings weniger flexibel, da die Varianz des Typen immer die gleiche bleibt und nicht an verschiedenen Stellen im Code variiert werden kann.

1.4 Echte Funktionstypen in Java-TX

Java-TX löst das Subtypisierung Problem durch die Einführung von echten Funktionstypen. Dadurch bekommen Lambda Ausdrücke echte Typen, welche nicht vom Kontext abhängig sind. Die Funktionstypen werden in Java-TX laut [25] mit der folgenden Syntax definiert:

$$(\tau_1, \tau_2, ..., \tau_N) \to \tau_0 \equiv FunN\$\$ < \tau_1, \tau_2, ..., \tau_N, \tau_0 >$$
$$(\tau_1, \tau_2, ..., \tau_N) \to void \equiv FunVoidN\$\$ < \tau_1, \tau_2, ..., \tau_N >$$

Außerdem sind Argumente der Funktionstypen automatisch kontravariant und der Rückgabewert kovariant. Dies erlaubt die Subtypisierung von Funktionstypen ohne use site Varianz, was zur besseren Lesbarkeit und weniger Fehleranfälligkeit führt. Tatsächlich ist die Verwendung von Wildcards für die Java-TX Funktionstypen nicht erlaubt [24][Abschnitt 6]. In Listing 1.19 ist ein Beispiel für die Subtypisierung von Funktionstypen in Java-TX zu sehen ist. Hier lässt sich die Funktion `func1` der Funktion `func2` zuweisen, da das Argument der Funktion `func1` ein Supertyp des Arguments der Funktion `func2` ist und der Rückgabewert der Funktion `fun1` ein Subtyp des Rückgabewerts der Funktion `fun2` ist, daher ist `fun1` nach 1.3.5 ein Subtyp von `fun2`. Dieser Code ist leichter lesbar

und verständlicher als der Java Code in Listing 1.18. Außerdem korrespondiert er direkt mit der theoretischen Grundlage in 1.3.5.

```
1  Fun1$$<Number, Integer> func1 = x -> x.intValue() + 1;
2  Fun1$$<Integer, Integer> func2 = func1;
```

Listing 1.19: Subtypisierung von Funktionstypen in Java-TX

Echte Funktionstypen ermöglichen in Java-TX die Definition von Funktionen in einem beliebigen Kontext. Ein Beispiel hierzu ist in Listing 1.20 zu sehen. In diesem Fall inferiert der Compiler den Typ `Fun2$$<Integer, Integer, Integer>`. In Java ist die Verwendung von `var` in diesem Kontext nicht möglich, wie in Listing 1.17 gezeigt wurde.

```
1 import java.lang.Integer;
2 public class Main{
3   main(){
4     var add = (x, y) -> x + y;
5   }
6 }
```

Listing 1.20: Lambda Ausdruck ohne Typkontext

1.5 GNU Make

GNU's Not Unix (GNU) Make ist ein Build-Management-Tool, welches von Richard Stallman und Roland McGrath entwickelt wurde [30][S.1]. Es wird hauptsächlich verwendet, um C Programme automatisiert zu kompilieren und zu linken. Make nutzt inkrementelle Kompilierung, d.h. es ermittelt mittels Timestamps welche Teile des Programms nach Änderungen des Quellcodes neu kompiliert werden müssen [30][S.5]. Grundsätzlich werden bei Make verschiedene Regeln in einem sogenannten Makefile definiert. Diese Regeln bestehen aus Zielen (targets), Abhängigkeiten (prerequisites) und Befehlen (recipe)[30][S.3]. Die Befehle werden ausgeführt, wenn die Abhängigkeiten neuer als die Ziele sind. Die Struktur einer Regel ist in Listing 1.21 zu sehen.

```
1  target ... : prerequisites ...
2    recipe
3    ...
4    ...
```

Listing 1.21: Aufbau einer Makefile-Regel aus [30][S.3]

Ein einfaches Makefile für ein C-Programm könnte wie das Beispiel in Listing 1.22 aussehen. Die Regeln in diesem Beispiel dienen zur Compilierung des Programms `edit`. Auf der

linken Seite einer Regel steht nun immer die Quelldatei, die erstellt werden soll. Auf der rechten Seite stehen die Abhängigkeiten, die für die Erstellung der Quelldatei benötigt werden. Die Befehle, die ausgeführt werden, wenn die Abhängigkeiten neuer sind als das Ziel, stehen unterhalb der Regel. In diesem Fall wird das Programm `edit` aus den Objektdateien `main.o, kbd.o, command.o, display.o, insert.o, search.o, files.o` und `utils.o` gebaut. Die Objektdateien werden aus den entsprechenden Quelldateien erstellt. Mit dem Befehl `make clean` können alle erstellten Dateien gelöscht werden. `clean` ist hierbei ein sogenanntes Phony Target. Das bedeutet, dass es sich nicht um eine Datei handelt. Vielmehr wird der definierte Befehl ausgeführt, sobald der Benutzer `make clean` aufruft [30][S.31].

```
edit : main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
            cc -o edit main.o kbd.o command.o display.o \
                      insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
.PHONY : clean
clean :
        rm edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Listing 1.22: Beispiel eines Makefiles aus [30][S.4]

GNU make bietet neben diesen grundlegenden make Funktionalitäten noch weitere mächtige Funktionen, z.B. die Möglichkeit Variablen zu definieren, shell Befehle innerhalb des Makefiles aufzurufen oder die Verwendung von Wildcards, um dynamische Makefiles zu erstellen. Für Hochsprachen wie Java werden meistens spezielle Build-Management-Tools

wie Maven oder Gradle verwendet [14]. Diese sind jedoch oft speziell auf diese Sprache entwickelt weniger flexibel als make.

1.6 Selbstkompilierende Compiler

Im Compilerbau gilt ein selbstkompilierender Compiler als Qualitätsmerkmal. Ein Compiler ist selbstkompilierend, wenn er in der Sprache geschrieben ist, die er kompiliert. Dies bedeutet, dass der Compiler in der Lage ist, seinen eigenen Quellcode zu kompilieren. In der Regel gibt es in jeder großen Programmiersprache mindestens einen Compiler, der selbstkompilierend ist. Bekannte Beispiel hierfür sind der javac Compiler, der in Java geschrieben ist [19] und Java Code kompiliert und der C++ Compiler der GNU Compiler Collection (GCC), der seit einigen Jahren in C++ geschrieben ist [7] und unter anderem C++ Code kompilieren kann [29][S.5]. Laut [32][Abschnitt 3.4] bieten selbstkompilierende Compiler im Wesentlichen 4 Vorteile:

- Es stellt einen nicht trivialen Test der Funktionsfähigkeit des Compilers dar.
- Sobald der selbstkompilierende Compiler einmal implementiert ist, ist die Entwicklung ohne die Abhängigkeit anderer Übersetzungssysteme möglich.
- Alle Verbesserungen, die am Backend des Compilers vorgenommen werden, wirken sich sowohl auf den Compiler, als auch auf den generierten Code aus.
- Es bietet eine umfassende Überprüfung der Selbstkonsistenz des Compilers. Der Compiler sollte in der Lage sein, seinen eigenen Quellcode zu kompilieren.

Der Prozess einen selbstkompilierenden Compiler zu erstellen, kann mit sogenannten T-Diagrammen visualisiert werden. In Abbildung 1.1 ist ein solches Diagramm für einen selbstkompilierenden Compiler in Java-TX zu sehen. Auf der linken eines T's ist dabei die Eingabesprache zu sehen, auf der rechten Seite die Ausgabesprache und in der Mitte die Sprache, in der der Compiler geschrieben ist. In diesem Fall ist das rote T der bereits bestehende Compiler in Java-TX, der den Java-TX Code in Bytecode übersetzt. Der blaue T ist das langfristige Ziels des Projekts „Java-TX Compiler in Java-TX", also ein Compiler der in Java-TX geschrieben ist und Java-TX Quelldateien in Bytecode kompiliert. Die Anordnung der T's symbolisiert, dass der ursprüngliche Compiler den neuen Compiler initial kompiliert.

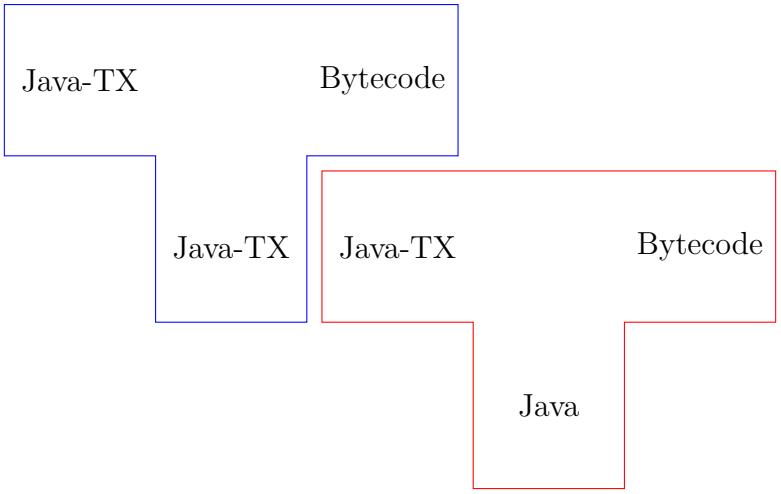

Abbildung 1.1: Selbstkompilierender Compiler in Java-TX

2 Aufbau der Umgebung

2.1 Voraussetzungen

Der aktuelle „Java-TX Compiler“ ist in Java implementiert. Da Java-TX ein Superset von Java 8[1] ist, kann der Quellcode im Wesentlichen übernommen werden. Es müssen lediglich die inferierbaren Typinformationen entfernt werden, um die Vorteile von Java-TX auch zu nutzen. Da sowohl Java, als auch Java-TX Java Bytecode generieren, ist es möglich, Java-TX und Java Dateien zu mischen. Dadurch kann der Compiler sukzessive in Java-TX übersetzt werden. Der Vorteil daran ist, dass man nach jeder übersetzten Datei einen funktionsfähigen Compiler hat, der zu einem gewissen Grad bereits aus Java-TX Dateien besteht. Auf diesem Zwischenstand können dann z.B. auch Tests aufgeführt werden, um die Funktionalität zu prüfen.

Ein Problem besteht darin, dass der „Java-TX Compiler“ `.jav`[2]-Dateien und `.class`-Dateien lesen kann, während der javac Compiler `.java`-Dateien und `.class`-Dateien einlesen kann. Jedoch kann der „Java-TX Compiler“ keine `.java`-Dateien lesen und der javac Compiler natürlich keine `.jav`-Dateien. Das bedeutet, dass zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien nicht ohne Weiteres möglich sind. Dieses Problem ist in Abbildung 2.1 visualisiert. Ein Pfeil symolisieren dabei Abhängigkeiten zu einer anderen Datei. Wenn der „Java-TX Compiler“ zuerst aufgerufen wird, kann er die Java Datei nicht lesen und umgekehrt. Selbst wenn die Abhängigkeiten nicht zirkulär sind, muss die genaue Reihenfolge der Kompilierung definiert sein. Dies bedeutet das der Build-Prozess für den „Java-TX Compiler in Java-TX“ sehr komplex werden würde.

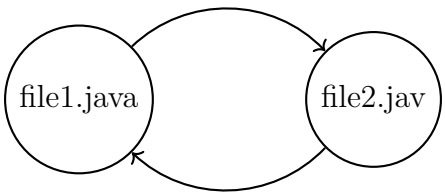

Abbildung 2.1: Zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien

Aktuell wird dieses Problem gelöst indem der gesamten Compiler zu Beginn einmal in seiner Ursprungsform (also nur `.java` Dateien) mit javac kompiliert wird. Dann liegt

[1] Es existieren einige Einschränkungen vgl. [25][S.2]
[2] Java-TX Quelldateien haben die Dateiendung `.jav`

der komplette „Java-TX Compiler“ in Form von `.class` Dateien vor. Anschließend muss diese Hierarchie nur noch als Classpath des „Java-TX Compiler“ angegeben werden. Der „Java-TX Compiler“ kann nun die `.class` Dateien lesen, die zur jeweiligen `.java` Datei korrespondieren. Wichtig ist an dieser Stelle natürlich, dass die `.class`-Dateien den gleichen Stand, wie die Quelldateien haben. Dieses Verhalten ist in Abbildung 2.2 gezeigt. Die Grafik zeigt, dass der „Java-TX Compiler“ zur Kompilierung der `.jav` Datei die zuvor generierte `.class` Datei von `file1` verwendet. Wenn später dann `file1` von javac kompiliert wird, wurde `file2.class` bereits vom „Java-TX Compiler“ erstellt. Der javac Compiler benötigt die vorkompilierten Klassen also nicht mehr im Classpath und kann die vom „Java-TX Compiler“ generierten .class Dateien verwenden. Natürlich ist es für diese Lösung notwendig, dass der „Java-TX Compiler“ zuerst alle `.jav` Dateien kompiliert, bevor der javac Compiler auf Basis der generierten `.class` Dateien, alle `.java` kompiliert. Diese Reihenfolge ist aber ohnehin notwendig, damit javac seinen Bytecode mit den korrekten Typen generieren kann, die der „Java-TX Compiler“ inferiert hat. Diese Lösung ist zwar nicht ideal, da man eine Abhängigkeit zum gesamten vorkompilierten Projekt benötigt. Da der „Java-TX Compiler“ aber keine `.java` Dateien lesen kann, ist es vermutlich die einzige Möglichkeit, zirkuläre Abhängigkeiten zwischen `.java` und `.jav` Dateien zu lösen.

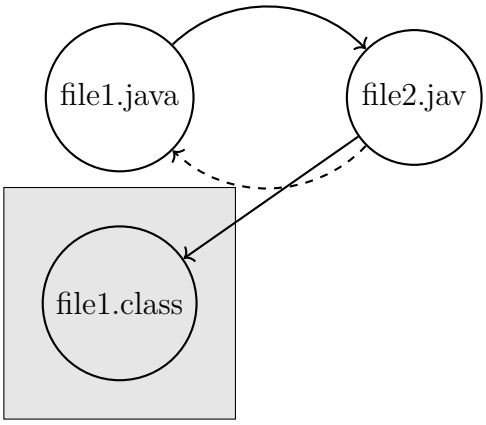

Abbildung 2.2: Zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien behoben

Die aktuelle „Java-TX Compiler“ Implementierung automatisiert den Build-Prozess mit Maven. Java ist für die Verwendung von Java basierten Sprachen entwickelt worden [16]. Da Maven keine Java-TX Unterstützung bietet, muss der Build-Prozess für den „Java-TX Compiler in Java-TX“ manuell angepasst werden. Dazu wurden zwei Lösungsansätze erarbeitet, die in den folgenden Abschnitten beschrieben werden.

2.2 Kompilierung mit Make

Der erste Ansatz der Verfolgt wurde, war die Kompilierung mittels GNU make (vgl. Abschnitt 1.5) umzusetzen. Make ist sehr flexibel und ist vielseitig einsetzbar. Um nicht alle Quelldateien einzeln angeben zu müssen, werden Wildcards verwendet. Außerdem bietet make den Vorteil, das nur geänderte Dateien neu kompiliert werden. Dies ist besonders bei großen Projekten von Vorteil, da nicht alle Dateien neu kompiliert werden müssen. Das verwendete Makefile ist in Listing 2.1 zu sehen.

```
1  JFLAGS = -g:none -implicit:none -nowarn
2  JC = javac
3  JTX = JavaTXcompiler-1.0-jar-with-dependencies.jar
4  SRCDIR = javatx-src/main/java
5  DESTDIR = out
6
7  # Use find to locate all .java and .jav files recursively
8  JAVASOURCES := $(shell find $(SRCDIR) -name '*.java')
9  JAVSOURCES := $(shell find $(SRCDIR) -name '*.jav')
10
11 # Convert .java files to .class files with the same directory
       structure
12 JAVACLASSES := $(patsubst $(SRCDIR)/%.java,$(DESTDIR)/%.class,$(
       JAVASOURCES))
13 JAVCLASSES := $(patsubst $(SRCDIR)/%.jav,$(DESTDIR)/%.class,$(
       JAVSOURCES))
14
15 default: $(JAVCLASSES) $(JAVACLASSES)
16
17 # Rule for compiling .jav files
18 $(DESTDIR)/%.class: $(SRCDIR)/%.jav
19     java -jar $(JTX) -d "$(DESTDIR)" -cp "$(SRCDIR):$(DESTDIR):
          target/dependencies/" $<
20
21 # Rule for compiling .java files
22 $(DESTDIR)/%.class: $(SRCDIR)/%.java
23     $(JC) -d $(DESTDIR) -cp "$(SRCDIR):$(DESTDIR):target/
          dependencies/*" $(JFLAGS) $<
24
25 .PHONY: clean
26 clean:
```

```
27      $(RM) -r $(DESTDIR)/*
```

Listing 2.1: Makefile für die Kompilierung des „Java-TX Compiler in Java-TX“

Im ersten Abschnitt werden einige globale Variablen, wie den Ort des javac und „Java-TX Compiler“ und der Quell- und Zielordner definiert. Dann werden mit dem shell Befehl `find` alle Dateien die auf .java und .jav enden in den Variablen `JAVASOURCES` und `JAVSOURCES` gespeichert. Der `find` Befehl geht dabei rekursiv vor und beachtet auch alle Unterverzeichnisse des Quellverzeichnisse. In diesem Fall findet der Befehl also alle Dateien in der Packethierarchie. Danach wird der `patsubst` Befehl von make verwendet, um den Pfand, an dem die korrespondierende `.class` Datei liegen wird, zu substituieren. So wird z.B. die Liste der Quelldateien

```
javatx-src/main/java/de/dhbwstuttgart/typeinference/assumptions/Assumption.java
javatx-src/main/java/de/dhbwstuttgart/typeinference/assumptions/FieldAssumption.java
```

zu einer Liste mit folgenden korrespondierenden Zieldateien

```
out/de/dhbwstuttgart/typeinference/assumptions/Assumption.class
out/de/dhbwstuttgart/typeinference/assumptions/FieldAssumption.class
```

Die genaue Syntax des `patsubst` Befehls kann in [30][S.92] nachgelesen werden. In der nächsten Zeile wird die Standardregel für das Makefile definiert. In GNU make ist die erste Regel diejenige, die aufgerufen wird, wenn der make Befehl ohne Parameter aufgerufen wird [30][S.109]. Diese Regel versucht alle Dateien in `$(JAVCLASSES)` und `$(JAVSOURCES)` zu kompilieren. Dazu werden die nächste beiden Regeln verwendet.

Die erste Regel kompiliert Java-TX Dateien. Dazu liegt der Compiler als JAR-Archiv vor. Der Compiler wird mit dem Befehl `java -jar` aufgerufen, welcher die Ausführung von JAR-Archiven ermöglicht. Der `-d` Parameter gibt den Zielordner an, in dem die `.class` Datei gespeichert wird. Der `-cp` Parameter gibt die Klassenpfade an, die der Compiler benötigt. In diesem Fall sind das der Quellordner, der Zielordner und die Abhängigkeiten des Projekts. Der `$<` Operator gibt den Pfad der ersten Abhängigkeit an [30][S.131]. In diesem Fall ist das die `.jav` Quelldatei. Das `%` Symbol dient als Wildcard und steht für eine beliebige Zeichenkette. Eine solche Regeln nennt man auch Musterregel (engl. Pattern Rule) [30][S.129].

Die zweite Regel kompiliert `.java` Quelldateien. Sie unterscheidet sich nur marginal von der ersten Regel. Lediglich der Aufruf des Compilers und die Dateiendung der Regel wurden angepasst, da hierzu der javac Compiler verwendet wird. Die Parameter `-d` und

`-cp` sind identisch. Lediglich einige weitere Parameter, die Anfags in die Variable `JFLAGS` gespeichert wurden, werden dem Compiler übergeben. Der Parameter |g:none| bedeutet, dass keine Debuginformationen generiert werden sollen. Das macht den generierten Bytecode in solch einem Entwicklungsumfeld leichter lesbar. Der Parameter |implicit:none| bedeutet, dass Abhängigkeit nicht impliziert kompiliert werden sollen. Weiteres dazu in Unterabschnitt 2.2.1. Der Parameter `nowarn` bedeutet schließlich, dass keine Warnungen ausgegeben werden sollen.

Die letzte Regel des Makefiles ist eine Phony Regel (vgl. Abschnitt 1.5). Sie löscht den Inhalt des Zielordners. Dazu wird der `rm` Befehl mit dem `-r` Parameter verwendet, um rekursiv alle Dateien und Ordner zu löschen. Der `*` Operator steht für alle Dateien und Ordner im Zielordner. Die Variable RM ist eine vordefinierte Variable in make, die den Befehl zum Löschen von Dateien definiert [30][S. 125 ff.]. Die Regel wird ausgeführt, wenn der make Befehl mit dem Parameter `clean` aufgerufen wird.

2.2.1 Performanceprobleme

Das Makefile ist funktionstüchtig und kompiliert das Projekt korrekt, allerdings geht eine komplette Kompilierung des Projekts mehrere Minuten, während die originale Java Version des Compilers nur wenige Sekunden benötigt. Dabei ist zu erwähnen, dass ein Großteil der Performanceeinbußen nicht durch die Typinferenz des „Java-TX Compiler“ entsteht, sondern schon der javac Compiler deutlich langsamer als beim ursprünglichen Compiler ist. Dies hängt damit zusammen, dass der javac Compiler für jede Datei einzeln aufgerufen wird. Dies resultierte vor allem auf Grund folgender zwei Gründe in einer schlechteren Performance:

- Der javac Compiler kompiliert standardmäßig alle Abhängigkeiten implizit mit. Da der Compiler aber sowieso für jede Datei aufgerufen wird, bedeutet das, dass viele Dateien mehrfach kompiliert werden.
- Der Overhead den Compiler zu starten ist relativ hoch. Das liegt zu großen Teilen wahrscheinlich daran, dass der javac Compiler in Java implementiert ist und somit die Java Virtual Machine (JVM) gestartet werden muss. Dadurch ist es sehr ineffizient den Compiler für jede Datei neu zu starten.

Die implizite Kompilierung kann beim Aufruf von javac durch den Parameter `-implicit:none` deaktiviert werden. Dieser sorgt dafür, dass Abhängigkeiten nicht implizit kompiliert werden. In Tabelle 2.1 sind die Kompilierzeiten des „Java-TX Compiler“ mit javac aufgeführt.

	Einzeln	Einzeln (Nicht Implizit)	Gemeinsam
Min	04:52,44	02:23,28	00:02,50
Max	05:10,69	02:25,78	00:02,73
Avg	04:58,83	02:24,15	00:02,57

Tabelle 2.1: Kompilierzeiten des „Java-TX Compiler“ mit javac[1]

Die erste Spalte beschreibt die Kompilierzeiten, wenn der Compiler für jede Datei einzeln aufgerufen wird und die implizite Kompilierung von Abhängigkeiten aktiviert ist. Die zweite Spalte beschreibt die Kompilierzeiten, wenn der Compiler für jede Datei einzeln aufgerufen wird, aber die Abhängigkeiten nicht implizit kompiliert werden. Es werden also mehrfache Kompilierungen vermieden. Die dritte Spalte beschreibt die Kompilierzeiten, wenn der Compiler für alle Dateien auf einmal aufgerufen wird. Zu sehen ist, dass sich die Kompilierzeit in etwa halbiert, wenn die Abhängigkeiten nicht implizit kompiliert werden. Allerdings ist die letzte Variante, die alle Dateien auf einmal an den Compiler übergibt, immer noch etwa um den Faktor 56 schneller.

Aus diesem Grund wurde entschieden, dass die Kompilierung des „Java-TX Compiler“ mit make nicht praktikabel ist. Ein alternativer Ansatz, der dieses Problem umgeht, wird in Abschnitt 2.3 vorgestellt.

2.3 Kompilierung mit Bash

Aufgrund der Laufzeiteinbusen, die beim Verwenden des Makefiles zustande kommen, musste eine alternative Lösung gefunden werden. Die Idee ist zuerst alle Dateien, die kompiliert werden müssen zu sammeln und dann den Compiler nur einmal aufzurufen. Da sich die Komplexität für solch ein Skript in Grenzen hält und die größtmögliche Flexibilität bietet, wurde sich für ein eigenes Skript entschieden. Als Skriptsprache wurde Bourne Again Shell (Bash) ausgewählt. Diese ist sowohl unter Linux, als auch unter MacOS nativ ohne weitere Abhängigkeiten lauffähig. Zur Ausführung auf Windows müsste man auf das Windows Subsystem for Linux (WSL) zurückgreifen. Das Skript hat das gleiche Verhalten wie das Makefile. Der einzige Unterschied besteht darin, dass alle Dateien die kompiliert werden müssen in eine Liste gespeichert werden und der Compiler am Ende nur einmal aufgerufen wird. Dadurch wird der Overhead des Compilers minimiert und die Kompilierzeit deutlich reduziert, sodass man ohne Java-TX Dateien mit dem Skript eine Zeit von etwa 2 Sekunden erreicht, was in etwa der Zeit in Tabelle 2.1 entspricht. Wenn das Skript mit dem Argument `clean` aufgerufen wird, wird der Inhalt des Zielordners gelöscht.

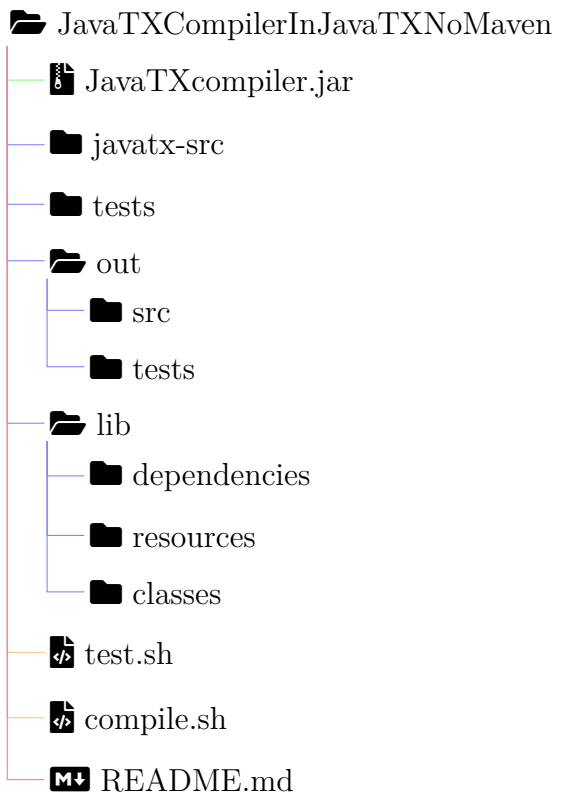

Abbildung 2.3: Dateistruktur des Projekts

In Abbildung 2.3 ist die Ordnerstruktur mit allen notwendigen Ressourcen des Projekts dargestellt. Das Projekt besteht aus folgenden Ordnern und Dateien:

- **JavaTXCompiler.jar**: Die aktuelle Version des „Java-TX Compiler“ in Form eines JAR-Archivs. Es wird verwendet, um die Java-TX Dateien des „Java-TX Compiler in Java-TX“ zu kompilieren.
- **javatx-src**: Enthält den Code des „Java-TX Compiler in Java-TX“. Besteht zum Großteil aus den Java Quelldateien des „Java-TX Compiler“ und einigen Java-TX Dateien, die bereits migriert werden konnten.
- **tests**: Enthält die Testsuite des „Java-TX Compiler“ in Form von JUnit Tests (vgl. Abschnitt 2.4).
- **out**: Enthält das kompilierte Projekt, sowie die kompilierten Tests.
  - **src**: Enthält das kompilierte Projekt in Form von `.class` Dateien.
  - **test**: Enthält die kompilierten Tests in Form von `.class` Dateien.
- **lib**: Enthält zusätzliche statische Dateien, die für das Projekt benötigt werden.

  - **dependencies**: Enthält externe Bibliotheken, die für das Projekt benötigt werden in Form von JAR-Archiven. Werden im „Java-TX Compiler“ durch Maven verwaltet.
  - **resources**: Enthält zusätzliche Ressourcen, die für die Tests benötigt werden. Im Wesentlichen sind dies die Beispielprogramme, die getestet werden.
  - **classes**: Enthält den gesamten „Java-TX Compiler“ in kompilierter Form um Abhängigkeiten zwischen `jav` und `java` Dateien zu ermöglichen (vgl. Abschnitt 2.1)

- **test.sh**: Dieses Bash Skript ist für die Kompilierung und Ausführung der Tests zuständig.
- **compile.sh**: Dieses Bash Skript ist für die Kompilierung des Projekts zuständig ist.
- **README.md**: Eine Markdown Datei, die Informationen zur Verwendung des Projekts enthält.

Da das gesamte Skript weder besonders spannend, noch komplex und zusätzlich relativ lang ist, wird es hier nicht im Detail beschrieben. Es ist jedoch im Anhang unter Abschnitt .1 zu finden.

2.4 Tests

Da es bei der Entwicklung eines eigenen Compilers schnell zu fehlerhaftem Bytecode kommen kann, welcher ggf. erst zur Laufzeit auffällt, ist es sinnvoll den generierten Bytecode zu testen. Der große Vorteil ist an dieser Stelle, dass es bereits eine Testsuite mit über 200 Unittests für den „Java-TX Compiler“ gibt. Diese Tests müssen lediglich auf den generierten Bytecode des „Java-TX Compiler in Java-TX“ angewandt werden, um fehlerhafter Bytecode oft frühzeitig zu erkennen.

Die Tests des „Java-TX Compiler“ werden über Maven verwaltet und ausgeführt. Außerdem bieten die meisten Integrated Development Environment (IDE)s eine direkte Integration für JUnit Tests. Diese automatischen Verfahren sind für reine Java Projekte geeignet, da sie die Testsuite automatisch ausführen und die Ergebnisse anzeigen. Da der „Java-TX Compiler in Java-TX“ aus einer gemischten Codebasis mit Java und Java-TX Quelldateien besteht, die mit dem eigenen Skript kompiliert wird, ist es nicht möglich diese Tools zu verwenden.

Daher muss die Testsuite manuell auf die kompilierten Dateien angewandt werden. Um diesen Prozess zu automatisieren wurde ein weiteres Skript geschrieben. Dieses Skript

kompiliert und kopiert die notwendigen Dateien zur Ausführung der Tests in den korrekten Ordner. Danach können die Tests mit einem Aufruf des JUnit 4 JAR-Archivs ausgeführt werden. Das gesamte Skript ist in Listing 2.2 zu sehen.

```
1  #!/bin/bash
2  ##TEST ENVIRONMENT##
3
4  DESTDIR="out/src"
5  TESTDESTDIR="out/tests"
6  DEPENDENCIES="dependencies/*"
7  TESTFILES="TestComplete TestPackages GenericParserTest
       TestTypeDeployment finiteClosure.SuperInterfacesTest astfactory
       .ASTFactoryTest targetast.ASTToTypedTargetAST targetast.
       GreaterEqualTest targetast.GreaterThanTest targetast.
       InheritTest2 targetast.InheritTest targetast.LessEqualTest
       targetast.LessThanTest targetast.OLTest targetast.PostIncTest
       targetast.PreIncTest targetast.PutTest targetast.TestCodegen
       targetast.TestGenerics targetast.TphTest targetast.WhileTest"
8  RESOURCES="lib/resources"
9
10 #recompile all necessary test files
11 javac -cp "$TESTDESTDIR:$DESTDIR:$DEPENDENCIES" -d $TESTDESTDIR
       tests/**/*.java
12 javac -cp "$TESTDESTDIR:$DESTDIR:$DEPENDENCIES" -d $TESTDESTDIR
       tests/*.java
13
14 cp -r $RESOURCES $TESTDESTDIR/resources/
15
16 cd "$TESTDESTDIR"
17
18 #run tests with junit
19 java -cp "../src:.:../../dependencies/*" org.junit.runner.
       JUnitCore $TESTFILES
```

Listing 2.2: Skript zum Kompilieren und Ausführen der Tests

Zu Beginn werden einige Variablen initialisiert. `DESTDIR` gibt den Ordner an, an dem die bereits kompilierten Dateien des „Java-TX Compiler in Java-TX“ liegen. Auf diesen Dateien werden die Tests ausgeführt. `TESTDESTDIR` gibt den Ordner an, in dem die kompilierten JUnit Testdateien abgelegt werden sollen. `DEPENDENCIES` gibt den Ordner mit den externen Abhängigkeiten des Projekts an. Diese sind identisch zu den Abhängigkeiten des `compile.sh` Skripts und damit die Dateien, die im ursprünglichen Compiler über Maven verwaltet werden. `TESTFILES` gibt die Testklassen an, die ausgeführt werden sollen. Zuletzt

verweist `RESOURCES` auf den Ordner, in dem die Testdateien, d.h. `.jav` Quelldateien, die von den Tests kompiliert werden, liegen. Dann werden alle JUnit Test Dateien kompiliert, wenn sie nicht bereits vorhanden sind. Im Anschluss werden sämtliche Ressourcen, die von den Tests benötigt werden, z.B. die Testdateien, deren Code kompiliert werden soll, an die korrekte Position kopiert, sodass die Tests diese zur Laufzeit finden können. In den nächsten Zeilen wird in den Testordner gewechselt und die Tests mit der Klasse `JUnitCore` ausgeführt. Die Klasse `org.juni.runner.JUnitCore` ist die Hauptklasse von JUnit 4, die die Tests ausführt. Sie ist im JUnit4 JAR-Archiv enthalten, welches bereits bei den Abhängigkeiten in `DEPENDENCIES` vorhanden ist.

3 Aufgetretene Probleme

Beim Versuch, im Zuge dieser Studienarbeit Teile des „Java-TX Compiler“ in Java-TX zu konvertieren, sind viele Bugs und fehlende Features aufgefallen. Einige interessante Probleme werden in diesem Kapitel genauer beschrieben. In Abbildung 3.1 ist eine Gesamtübersicht über die Anzahl der gefundenen Probleme zu sehen. Die Grafik ist in zwei Kategorien unterteilt: Bugs und Feature Anfragen. Bugs beschreiben hierbei Fehler in bereits implementierten Funktionen, während Feature Anfragen gänzlich neue Funktionen beschreiben, meistens handelt es sich hierbei um Standard Java Funktionen, die bislang noch nicht implementiert wurden und keine neuen Funktionen im Bezug auf Java-TX Sprachfeatures. Weiter sind die Kategorien nach offenen und geschlossenen Problemen unterteilt.

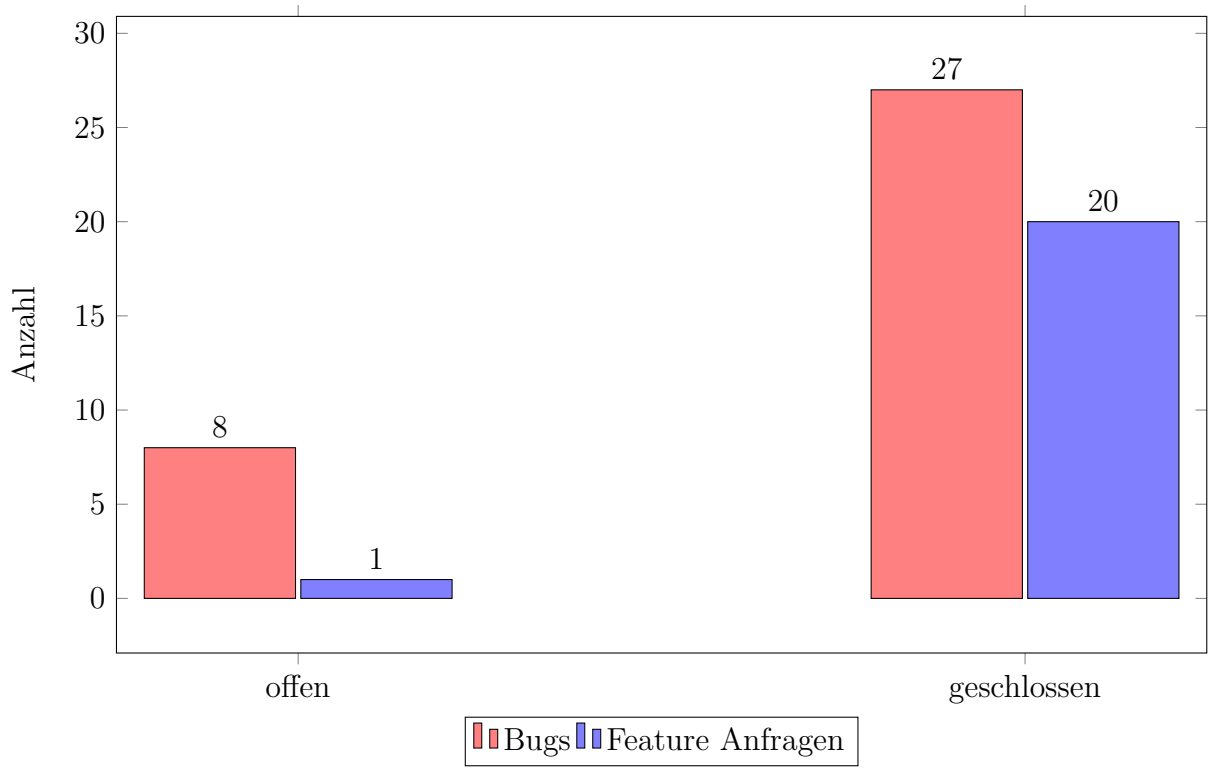

Abbildung 3.1: Gefundene Probleme und neue Funktionen

3.1 Neue Funktionen

Um den „Java-TX Compiler in Java-TX“ umzuschreiben sind einige Funktionen notwendig, die zu Beginn der Studienarbeit noch nicht implementiert waren. Über den Zeitraum der Studienarbeit wurden insgesamt 20 neue Funktionen implementiert. Da die For-Each Schleife in Java-TX minimal von der Java Syntax abweichen darf, wird sie im folgenden etwas ausführlicher beschrieben. Einige der anderen Funktionen werden nur kurz aufgezählt.

3.1.1 ForEach Schleife

Die For-Each Schleife, ermöglicht es mit einer eleganten Syntax über eine Variable vom Typ `java.lang.Iterable` zu iterieren [33]. Ein Beispiel zur Verwendung der For-Each Schleife in Java ist in Listing 3.1 zu sehen.

```
1    List<String> list = new ArrayList<>();
2    //entweder
3    for (String s : list) {
4        System.out.println(s);
5    }
6    // ... oder
7    for(var s : list) {
8        System.out.println(s);
9    }
```

Listing 3.1: For-Each Schleife in Java

Dabei kann der Typ der Variable entweder explizit angegeben, oder durch Angabe des Schlüsselwort `var` durch den Compiler inferiert werden. In Java-TX kann diese Typangabe Angabe auch gänzlich weggelassen werden. Ein Beispiel zur Verwendung der For-Each Schleife in Java-TX ist in Listing 3.2 zu sehen.

```
1    List<String> list = new ArrayList<>();
2    //Kein Typ vor der Variable s notwendig
3    for (s : list) {
4        System.out.println(s);
5    }
```

Listing 3.2: For-Each Schleife in Java-TX

3.1.2 Weitere neue Funktionen

Neben der For-Each Schleife wurden noch weitere Funktionen implementiert. Diese unterscheiden sich allerdings nicht von der Java Syntax und werden daher nur kurz aufgelistet:

- super() Konstruktoraufruf
- Methodenaufruf der Superklasse mittels super
- this() Konstruktoraufruf
- instanceof Schlüsselwort
- throw Schlüsselwort
- Verwendung des `null` Literals
- `Character` Literal z.B. `'a'`
- `Long` und `Float` Literals z.B. `1L` oder `1.0f`
- Type Casts
- Negation Operator `!`
- Bitweise AND `&` und OR `|` Operatoren
- Modulo Operator `%`
- Ternary Operator `? :`
- Do-While Schleife
- Verwenden von JAR-Archiven im Classpath

3.2 Bugs

Insgesamt wurden 27 Bugs gefunden und behoben, einige weitere sind noch offen. Im folgenden Abschnitt wird eine kleine Auswahl dieser Bugs ausführlicher beschrieben.

3.2.1 JVM Classpath wird von „Java-TX Compiler“ beachtet

Zur Beschreibung dieses Bugs müssen zunächst einige Grundlagen zum Classpath und dem Classloader geklärt werden. Damit der „Java-TX Compiler“ andere Klassen verwenden kann (darunter zählen z.B. auch JAR-Archive), müssen sie im Classpath vorhanden oder Teil der Standardbibliothek sein. Der Classpath kann, wie auch beim javac Compiler, mit dem Flag `-cp` oder `-classpath` angegeben werden. Ein Beispielaufruf könnte folgendermaßen aussehen: `java -jar JavaTXCompiler.jar -cp /path/to/classes /path/to/source.jav`[1]. Wenn kein Classpath angegeben wird, wird das aktuelle Arbeitsverzeichnis des Nutzers verwendet. Die angegebenen Pfade werden dann vom Compiler durchsucht, um die benötigten Klassen zu finden. Wenn die Klasse nicht gefunden werden konnte, wird ein Fehler ausgegeben. Nur Klassen, die im Classpath vorhanden sind, können importiert und verwendet werden. Ausnahme sind die Klassen aus der Java Standard Library, wie z.B. alle Klassen des Pakets `java.lang`, die standardmäßig vorhanden sind. Das Packet `java.lang` bietet die Klasse `ClassLoader`, die es ermöglicht, Klassen dynamisch in die JVM zu laden [1]. Der „Java-TX Compiler“ verwendet auch diesen Classloader, um die angegebenen Klassen zu suchen und einzulesen. So muss kein eigener Parser für Java Bytecode implementiert werden.

Um genau zu sein gibt es nicht einen Classloader, sondern mehrere, die nacheinander ausgeführt werden. Die oberste Schicht ist der Bootstrap Classloader, der Java Development Kit (JDK) interne Klassen lädt. Darunter befindet sich der Extension Classloader, der die Klassen aus den JAR-Dateien im `jre/lib/ext` Verzeichnis lädt. Der Application Classloader lädt die Klassen aus dem Classpath und zuletzt gibt es eine eigene Implementierung des `java.net.URLClassLoader` namens Directory Classloader, der die Pfade, die dem Compiler mit `-cp` übergeben werden, durchsucht [1, 18]. Eine Visualisierung dieser Schichten ist in Abbildung 3.2 zu sehen. Das Programm beginnt mit dem untersten Classloader (Directory Classloader), versucht also die gesuchte Klasse in diesen Ressourcen zu finden. Wenn die Klasse nicht gefunden wird, wird der nächste Classloader (Application Classloader) verwendet. Dieser Prozess wird so lange wiederholt, bis die Klasse gefunden wird oder alle Classloader durchsucht wurden. Zu beachten ist, dass der Classpath des Application Classloaders nicht derselbe ist, wie der Classpath, der dem Compiler übergeben wird und vom DirectoryClassloader verwendet wird. Der Classpath des Application Classloaders ist der Classpath, mit dem die JVM gestartet wurde. Dieser beinhaltet in diesem Fall z.B. sämtliche Maven Abhängigkeiten und Bytecode Dateien des „Java-TX Compiler“.

[1] JavaTXCompiler.jar ist in diesem Fall der „Java-TX Compiler“ in Form eines JAR-Archivs

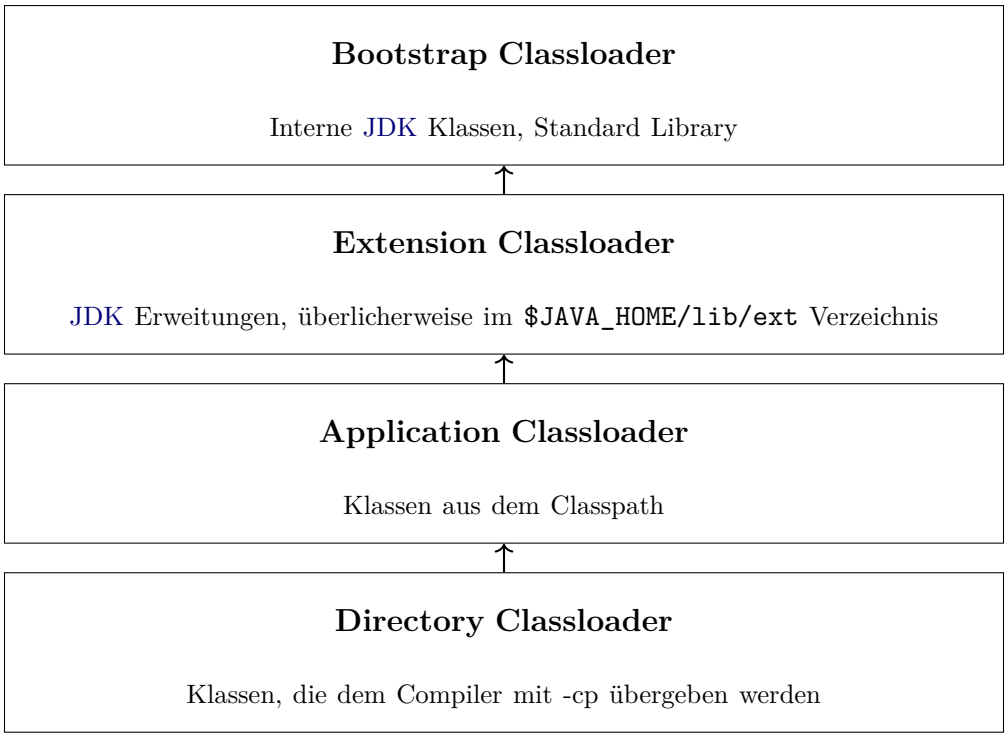

Abbildung 3.2: Die Classloader Hierarchie des „Java-TX Compiler“ (vgl. [18])

Da durch dieses Vorgehen auch die Klassen aus dem JVM Classpath vom Classloader und damit vom „Java-TX Compiler“ berücksichtigt werden, kann man diese importieren, ohne sie im Classpath angegeben zu müssen. Dies ist ein unerwünschtes Verhalten, da der Compiler so Klassen akzeptiert, die gegebenenfalls nicht vom Programmierer gewünscht sind, was zu Verwirrung führen kann. Ein Beispiel hierzu ist in Listing 3.3 zu sehen. Dieser Code kompiliert mit dem Befehl `java -jar JavaTXCompiler.jar Main.jav`[1] korrekt, obwohl die Klasse `com.google.common.math.IntMath` weder im Classpath angegeben ist, noch in der Standardbibliothek vorhanden ist. Sie ist jedoch im JAR Archiv des „Java-TX Compiler“ und damit im JVM Classpath zur Ausführung des Compilers vorhanden, da der „Java-TX Compiler“ die Google Guava Bibliothek verwendet.

```
1 import com.google.common.math.IntMath;
2 import java.lang.String;
3
4 class Main{
5     return2(){
6         return IntMath.checkedAdd(1, 1);
7     }
8 }
```

Listing 3.3: Verwenden von Klassen im JVM Classpath

[1] JavaTXCompiler.jar ist hier der „Java-TX Compiler“ in Form eines JAR Archivs

Dieses Verhalten ist vor allem auch für das Projekt „Java-TX Compiler in Java-TX“ problematisch, da der „Java-TX Compiler“ und der „Java-TX Compiler in Java-TX“ die gleichen Klassen/Klassenhierarchie verwenden und somit die Möglichkeit besteht, dass der Compiler die Klassen des „Java-TX Compiler“ verwendet, anstatt die des „Java-TX Compiler in Java-TX“.

Die Lösung dieses Problems ist glücklicherweise unkompliziert. Es muss lediglich die ClassLoader Hierarchie so angepasst werden, dass der Application Classloader übersprungen wird. Diese Änderung ist in Abbildung 3.3 zu sehen. Der Directory Classloader ruft nun direkt den Extension Classloader auf. So können weiterhin Klassen aus der Java Standard Bibliothek verwendet werden, jedoch nicht mehr die Klassen im JVM Classpath.

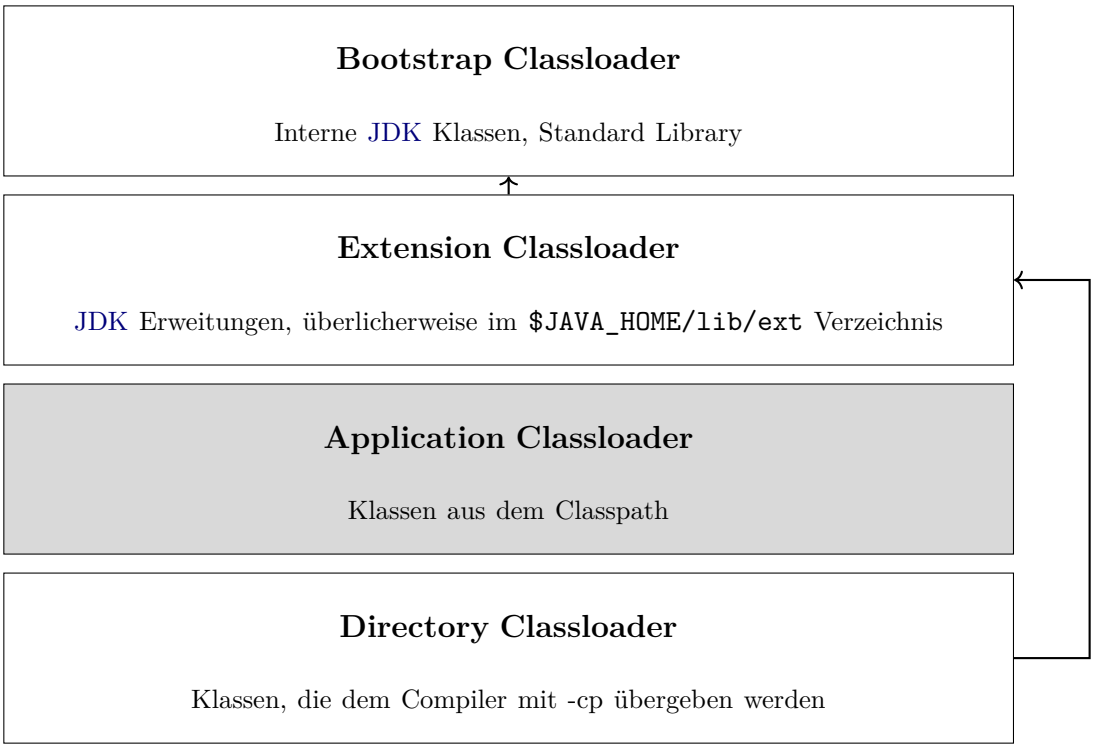

Abbildung 3.3: Die Classloader Hierarchie des „Java-TX Compiler“ ohne den ApplicationClassLoader

3.2.2 Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces

Das Problem beschreibt die Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces, die seit Java 8 als Zieltypen für Lambda Ausdrücken dienen (vgl. Unterabschnitt 1.3.1). Ziel ist es, bestehende Java Bibliotheken, die mit funktionalen Interfaces arbeiten, mit Java-TX Funktionstypen verwenden zu können. Die theoretische Lösung für dieses Problem wurde bereits 2017 in [24][Abschnitt 6] beschrieben. Die praktische

Umsetzung gestaltet sich jedoch komplizierter als gedacht. Ein Beispiel für eine Bibliothek, die ausgiebig funktionale Interfaces verwendet ist die sehr verbreitete Stream API, welche mit Java 8 eingeführt wurde [2]. Streams erlauben es Daten mit deklarativem Code zu verarbeiten, was die Lesbarkeit und Wartbarkeit des Codes erhöht [27].

```
1 import java.util.List;
2 import java.util.stream.Stream;
3 import java.util.function.Predicate;
4
5 public class ListUtils{
6     static List<Integer> getAllEvenNumbers(List<Integer> list){
7         List<Integer> result = list.stream().filter(x -> x % 2 ==
              0).toList();
8         return result;
9     }
10 }
```

Listing 3.4: Verwendung der Stream API in Java

In Listing 3.4 ist ein Beispielprogramm in Java zu sehen, welches die Stream API verwendet. Die Methode `getAllEvenNumbers` filtert alle ungeraden Zahlen aus einer Liste. Dazu verwendet sie die Lambda Funktion `x -> x % 2 == 0`, die folgendermaßen definiert ist:

$$\text{isEven}(n)^1 = \begin{cases} true, & \text{wenn } n \mod 2 = 0 \\ false, & \text{sonst} \end{cases}$$

Die `filter` Methode von `java.util.stream.Stream` lässt nur Werte passieren, für die die angegebene Funktion zu „true“ evaluiert. Zuletzt werden diese Werte in einer Liste gesammelt und zurückgegeben.

Der javac Compiler inferiert für den Lambda Ausdruck `x -> x % 2 == 0` den Typ `java.util.function.Predicate<Integer>`, da explizit dieser Typ von der `filter` Methode erwartet wird.

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {
    Stream<T> filter(Predicate<? super T> predicate);
    ...
}
```

[1] Im Quellcode hat der Lambda Ausdruck keinen Namen

In Java-TX kompilierte dieser Code zu Beginn der Studienarbeit nicht, da der „Java-TX Compiler“ den Typ `Fun1$$<Integer, Boolean>` für den Lambda Ausdruck inferiert hat. Der „Java-TX Compiler“ würde also den Lambda Ausdruck als Funktionstypen interpretieren, anstatt als funktionalen Interface, was soweit ja korrekt ist. Obwohl die beiden Typen semantisch äquivalent sind, sind sie auf Grund des nominalen Typsystems von Java und der JVM jedoch nicht austauschbar. Es kommt also zu einem Laufzeitfehler. Dieses Problem ist von hoher Relevanz, da sämtliche Methoden aus Java Bibliotheken, die funktionalen Interfaces verwenden, so in Java-TX nicht verwendet werden können.

Im Laufe der Arbeit wurde zumindest eine Teillösung dieses Problems implementiert, sodass der Code in Listing 3.4 kompiliert und lauffähig ist. Der „Java-TX Compiler“ generiert in solch einem Fall nun den Code für das korrekte funktionale Interface statt dem `FunN$$` Typen. Das funktioniert allerdings nur, wenn der Lambda Ausdruck wie bei Listing 3.4 direkt im Funktionsaufruf steht. Der Beispielcode in Listing 3.5 funktioniert also nicht und stellt aktuell noch ein Problem dar.

```
1  import java.util.List;
2  import java.lang.Integer;
3  import java.lang.Boolean;
4  import java.util.stream.Stream;
5  import java.util.function.Predicate;
6
7  public class ListUtils{
8      static getAllEvenNumbers(list){
9          // Java-TX inferiert hier Fun1$$<Integer, Boolean>
10         var func = x -> x % 2 == 0;
11         // An dieser Stelle wird aber ein Predicate<Integer>
               erwartet -> Laufzeitfehler
12         var result = list.stream().filter(func).toList();
13     }
14 }
```

Listing 3.5: Aktuell nicht lauffähiger Java-TX Code I

Das liegt daran, dass der Compiler zum Zeitpunkt der Initialisierung des Lambda Ausdrucks nicht weiß, in welchem Kontext der Ausdruck später verwendet werden soll. Betrachten wir zur Verdeutlichung des Problems die etwas komplexere Funktion `uselessFunction` in Listing 3.6. Über die Sinnhaftigkeit dieses Codes lässt sich streiten. Die Funktion sollte im Entdefekt das gleiche Resultat wie die Funktion in Listing 3.4, mit dem Unterschied das in der Resultatliste für jede Zahl der Wert `true` zurückgegeben wird, liefern. Das Problem ist nun, dass die selbe Lambda Funktion im Laufe der Funktion mit verschiedenen funktionalen Interfaces verwendet wird, was die Sache verkompliziert. So würde die `map`

Methode in Zeile 10 den Typ `Function<Integer, Boolean>` erwarten, während die Filter Methode in Zeile 9 weiterhin `Predicate<Integer>` erwartet. Hier müsste vermutlich für alle Aufrufe eine separate Lambda Funktion mit dem richtigen Target Type im Bytecode erstellt werden. Allerdings erfordert dies vermutlich eine größere Änderung am Compiler und konnte aktuell noch nicht umgesetzt werden.

```
1  import java.util.List;
2  import java.lang.Integer;
3  import java.lang.Boolean;
4  import java.util.stream.Stream;
5  import java.util.function.Predicate;
6  import java.util.function.Function;
7
8  public class ListUtils{
9      static uselessFunction(list){
10         // Java-TX inferiert hier Fun1$$<Integer, Boolean>
11         var func = x -> x % 2 == 0;
12         // An dieser Stelle wird ein Predicate<Integer> erwartet
13         var result1 = list.stream().filter(func);
14         // An dieser Stelle wird ein Function<Integer, Boolean>
               erwartet
15         var result2 = result1.map(func).toList();
16         return result2;
17     }
18 }
```

Listing 3.6: Aktuell nicht lauffähiger Java-TX Code II

3.2.3 Überschreiben von Methoden mit primitiven Datentypen

Ein weiteres Problem, ist das Überschreiben von Methoden mit primitiven Datentypen. Im Vergleich zu Java unterstützt Java-TX nur Referenztypen, keine primitiven Datentypen. Die Grammatik erlaubt zwar die Verwendung von primitiven Datentypen, diese werden jedoch intern in Referenztypen umgewandelt. In Abbildung 3.4 sind auf der linken Seite einige Initialisierungen von primitiven Datentypen in Java-TX gezeigt. Auf der rechten Seite gegenübergestellt ist der Code, wie er im Compiler intern verarbeitet und später auch im Bytecode generiert wird. Der Compiler generiert also für jeden primitiven Datentypen den entsprechenden Wrappertyp. Daher ist es auch bei der Verwendung von primitiven Typen notwendig die entsprechenden Wrapperklasse zu importieren.

```
1 import java.lang.Integer;
2 import java.lang.Boolean;
3 import java.lang.Float;
4 class PrimitiveTypes{
5     int i = 5;
6     boolean b = true;
7     float f = 10.5f;
8 }
```

```
1 import java.lang.Integer;
2 import java.lang.Boolean;
3 import java.lang.Float;
4 class PrimitiveTypes{
5     Integer i = 5;
6     Boolean b = true;
7     Float f = 10.5f;
8 }
```

Abbildung 3.4: Primitive Datentypen in Java-TX

Diese Eigenschaft von Java-TX führte im Zusammenhang mit der Überladung von Java Methoden, deren Rückgabewert oder Parameter primitive Datentypen sind, zu einem Problem. Nehmen wir als Beispiel den Code in Listing 3.7. Die Methode `hashCode` wird von der Mutterklasse `Object`[1] geerbt. Sie hat folgende Signatur:

```
int hashCode();
```

Da sowohl der Name, als auch die Parameterliste der Methode übereinstimmt, würde man erwarten, dass diese von der Klasse `Foo` überschrieben wird.

```
1 import java.lang.Integer;
2
3 public class Foo{
4     public hashCode(){
5         return 42;
6     }
7 }
```

Listing 3.7: Überschreiben von Methoden mit primitiven Datentypen in Java-TX

Stattdessen inferiert der Compiler allerdings die Typen in Listing 3.8 für die Methode `hashCode` in Listing 3.7. Dies ist Aufgrund der Tatsache, dass primitive Datentypen in Java-TX automatisch mit dem Wrappertyp ersetzt werden, logisch.

```
1 import java.lang.Integer;
2
3 public class Foo{
4     public java.lang.Integer hashCode(){
5         return 42;
6     }
7 }
```

Listing 3.8: Ergebnis der Typinferenz für die Methode `hashCode` in Java-TX

[1] In Java erben alle Klassen implizit von Object

Im Bytecode führt dies allerdings anstatt einer Überschreibung zu einer Überladung der Methode `hashCode`, da der Rückgabetyp in Listing 3.8 nicht mit dem Rückgabetyp in Listing 3.7 übereinstimmt, schließlich sind `int` und `java.lang.Integer` trotz Autoboxing unterschiedliche Typen (mehr dazu in [17][S.6 ff]). Denn obwohl Java die Überladung von Methoden anhand des Rückgabetyps nicht unterstützt, ist es in Java Bytecode durchaus möglich. Dies macht sich Java z.B. für das kovariante Überladen von Methoden zunutze. Seit Java 5 ist es möglich, dass eine Methode in einer Subklasse einen Rückgabetyp hat, der ein Subtyp des Rückgabetyps der Methode in der Superklasse ist [17][S. 49]. Dazu sei zunächst die Signatur der Methode `clone` in der Klasse `Object` gegeben, welche kein Parameter hat und ein Objekt vom Typ `Object` zurückgibt:

```
class Object{
    ...
    protected Object clone(){...}
}
```

Es ist nun möglich, die Methode `clone` in einer Subklasse zu überschreiben und den Rückgabetyp zu spezialisieren. In Listing 3.9 wird die Methode `clone` in der Klasse `A` überschrieben und der Rückgabetyp auf `A` spezialisiert.

```
1 class A{
2     ...
3     @Override
4     public A clone(){...}
5 }
```

Listing 3.9: Kovariante Methodenüberladung in Java

Dies wird durch eine Bridge Methode ermöglicht, die sich den Fakt zunutze macht, dass die JVM Methoden anhand des Rückgabetyps unterscheidet und somit überladen kann. Der Compiler erzeugt also eine Bridge Methode, mit der Signatur `Object clone()`, die die Methode `A clone()` aufruft. In Listing 3.10 ist dazu der dekompilierte Bytecode der Klasse `A` gegeben.

```
class A{
    ...
    public A clone(){...}

    public Object clone(){
        return this.clone(); //Aufruf der Methode clone():A
    }
}
```

Listing 3.10: Dekompilierter Bytecode der Klasse A

In unserem Bug ist dies dies allerdings nicht das gewünschte Verhalten, da eine Überladung anhand des Rückgabewerts nicht möglich sein sollte. In Java werden Überladungen anhand des Rückgabewerts vom Compiler abgelehnt, da nicht immer ersichtlich ist, welche Methode aufgerufen werden soll. Außerdem wären Überladungen von Methoden welche primitive Typen verwenden in Java-TX gänzlich unmöglich, weil selbst die explizite Angabe des Typs `int` vom Compiler in `java.lang.Integer` umgewandelt werden würde.

Um dieses Problem zu lösen wird aktuell eine einfache Substitution verwendet. Wenn der Compiler eine Überladung erkennt und der Rückgabetyp oder ein Parametertyp der Superklasse ein primitiver Datentyp ist, wird der dazugehörige Wrappertyp durch den jeweiligen primitiven Typen ersetzt. Dadurch funktioniert die Überschreibung von Methoden mit primitiven Datentypen korrekt. Da es aber noch einige Bugs mit dieser Implementierung gibt, bleibt abzuwarten, ob dies die endgültige Lösung ist.

3.2.4 Korrekter Methodenaufruf für überladene Methoden mit Subtypen als Parameter

Dieser Bug ist zum Zeitpunkt der Abgabe dieser Arbeit noch nicht behoben. Er tritt vor allem im Zusammenhang mit dem Visitor-Pattern auf, welches im „Java-TX Compiler“ ausgiebig benutzt wird. Der Compiler ruft nicht immer die korrekte Methode auf, wenn mehrere potenziell korrekte Methoden zur Auswahl stehen. Sehen wir uns dazu an, wie Java mit diesem Problem umgeht. Dazu sei der Code in Listing 3.11 gegeben. Die Main Funktion ruft dabei die Methode `visit` der Klasse `Visitor` mit einer Instanz der Klasse `java.lang.Integer` (bzw. int, was aber geboxed wird [17][S.6 ff]) auf. Die `visit` Methode ist dreimal überladen, einmal mit `java.lang.Object`, einmal mit `java.lang.Number` und einmal mit einem `java.lang.Integer`. Die Frage ist nun, welche dieser Überladungen aufgerufen werden sollte. Theoretisch wäre jeder Aufruf korrekt, da `java.lang.Integer` sowohl von `java.lang.Number` als auch von `java.lang.Object` erbt. Das Verhalten in so einem Fall ist in [10][Abschnitt 15.12.2.5] beschrieben. Java wählt in einem solchen Fall

die spezifischste Methode, also die Methode, die den spezifischsten Typen als Parameter hat. In diesem Fall wäre das die Methode, mit der Signatur `void visit(Integer i)`. Dies bestätigt sich auch, wenn der Code in Listing 3.11 kompiliert und ausgeführt wird. Die Ausgabe ist wie erwartet `"Integer"`.

```
class Main{
    public static void main(String[] args){
        Visitor v = new Visitor();
        v.visit(1);
    }
}

class Visitor{
    public void visit(Object o){
        System.out.println("Object");
    }
    public void visit(Number n){
        System.out.println("Number");
    }
    public void visit(Integer i){
        System.out.println("Integer");
    }
}
```

Listing 3.11: Überladene Methoden in Java

Dieses Verhalten wäre vor allem aus Kompatibilitätsgründen auch in Java-TX wünschenswert. Hier kommt es aktuell aber zu einem Fehler. Wenn der Code mehrmals kompiliert wird, wählt der Compiler jedes Mal eine andere Methode. Die Ausgabe ist also nicht deterministisch. Dies ist dadurch zu ergründen, dass der Typinferenzalgorithmus alle 3 Lösungen für den Methodenaufruf findet, aktuell aber nicht berücksichtigt, dass die Parameter der Methoden Subtypen sein können. Die Lösungen werden also als gleichwertig angesehen. Der Bytecodegenerator wählt dann aktuell die erste Lösung und verwirft die restlichen. Da der Typinferenzalgorithmus auf mehreren Threads parallel ausgeführt wird, kann sich die Reihenfolge der Ergebnisse ändern und somit auch das Ergebnis des Bytecodegenerators. Daher werden bei mehreren Kompilierungen unterschiedliche Ergebnisse erzielt.

Leider konnte das Problem bisher nicht gelöst werden, da es erst relativ spät bemerkt wurde. In einer späteren Version des Compilers sollte sich Java-TX in diesem Punkt aber wie Java verhalten, so dass auch das Visitor Pattern in Java-TX korrekt funktioniert.

3.2.5 Weitere Bugs und fehlende Features

Neben diesen umfangreich beschriebenen Problemen, gab es noch viele weitere Probleme, die hier nur kurz aufgeführt werden.

1. Die Access Modifier wurden nicht korrekt auf die Methoden angewendet. Alle Methoden wurden mit dem Access Modifier `public` deklariert. Wenn ein anderer Access Modifier verwendet wurde, wurden dieser einfach hinzugefügt. So konnte es zu Signaturen wie `public private void foo()` kommen.
2. Die Fehlermeldungen des Compilers wurden verbessert. Es wird nun angezeigt in welcher Zeile und Datei ein Constraint erstellt wurde.
3. Überladene Konstruktoren konnten nicht aufgerufen werden.
4. If-Statements ohne Blöcke wurden teilweise nicht korrekt verarbeitet.
5. `toString()` und andere Methoden von `Object` konnten auf Interfaces nicht aufgerufen werden.
6. Die @Override Annotation bei Methoden führte zu einem Fehler. Annotations wird nun ignoriert.
7. Die Typinferenz war für die neu hinzugefügte ForEach Schleife fehlerhaft.
8. Es gab einige Probleme mit Interfaces, welche behoben wurden.
9. Der Bytecode beim Aufruf von statischen Methoden war fehlerhaft.
10. Bei der Überschreibung von vererbten Methoden mussten die Parametertypen den gleichen Namen haben.

Zusätzlich sind auch einige Features aufgefallen, die bis zum aktuellen Stand noch nicht implementiert wurden.

1. Arrays und damit auch die Main Funktion sind nicht implementiert
2. Exceptions sind nur sehr rudimentär implementiert, z.B. sind checked Exceptions aktuell nicht möglich
3. Subtypisierung bei Funktionstypen funktioniert noch nicht wie es sollte
4. Der Funktionstyp FunVoidN$$ für Funktionstypen die void zurückgeben ist aktuell noch nicht umgesetzt

4 Vorteile in der Praxis

Generell inferiert Java-TX immer den generellsten Typ oder Prinzipaltyp. Für eine formale Definition des Prinzipaltyps in Java-TX wird an dieser Stelle auf [25][Abschnitt 5] verwiesen. Für Funktionstypen gilt dabei, dass der Definitionsbereich, also die Parameter maximal (möglichst generell) und der Wertebereich, also der Rückgabewert minimal (möglichst speziell) sein muss [25][Abschnitt 2]. Durch den Typinferenzalgorithmus kann daher daher ein generellerer Typ inferiert werden, als ein Programmierer angegeben hätte. Dadurch wird der Definitionsbereich der Funktion größer und die Wiederverwendbarkeit des Codes steigt potenziell. Zur Verdeutlichung ist in Listing 4.1 ein reales Beispiel aus dem Code des „Java-TX Compiler“ gegeben[1]. Die Aufgabe der Klasse ist in diesem Fall nicht wichtig, daher wurde der Quellcode in diesem Beispiel auf die relevante Methode reduziert. Wie man sehen kann, erwartet die Methode als Parameter eine Liste des Typs `GenericRefType`, über welche dann mittels For-Each Schleife iteriert wird, wobei die einzelnen Elemente genutzt werden, um eine neue Liste zu erstellen. Die Typisierung des Parameters ist dabei keineswegs ungewöhnlich und wäre vermutlich von vielen Programmieren so oder so ähnlich gewählt worden. Aber handelt es sich bei der Signatur hierbei um die optimale/generellste Typisierung?

```
1  public class FunNClass extends ClassOrInterface {
2      private static GenericDeclarationList createGenerics(List<
           GenericRefType> funNParams) {
3          List<GenericTypeVar> generics = new ArrayList<>();
4          for (GenericRefType param : funNParams) {
5              generics.add(new GenericTypeVar(param.getParsedName()
                   ,
6                      new ArrayList<>(), new NullToken(), new
                           NullToken()));
7          }
8          return new GenericDeclarationList(generics, new NullToken
               ());
9      }
10     ...
```

[1] Die Imports der Klasse wurden bei den Beispielen in diesem Abschnitt bewusst weggelassen, um den Quellcode kompakt zu halten. Es gilt aber natürlich, dass alle verwendeten Typen importiert werden müssen (vgl. Abschnitt 1.2)

```
11 }
```

Listing 4.1: Beispielklasse aus dem „Java-TX Compiler“

Sehen wir und dazu das Beispiel in Listing 4.2 an.

```
1  public class FunNClass extends ClassOrInterface {
2      private static createGenerics(funNParams) {
3          var generics = new ArrayList<>();
4          for (param : funNParams) {
5              generics.add(new GenericTypeVar(param.getParsedName()
                   ,
6                       new ArrayList<>(), new NullToken(), new
                           NullToken()));
7          }
8          return new GenericDeclarationList(generics, new NullToken
               ());
9      }
10     ...
11 }
```

Listing 4.2: Listing 4.1 ohne Typinformationen

Hier wurden alle inferierbaren Typen entfernt. Relevant ist hier vor allem der Rückgabetyp und der Typ des Parameters der Methode. Außerdem muss der generische Typ der Liste `generics` nicht mehr explizit angegeben werden und im Kopf der For-Each Schleife wurde der Typ der Variablen `param` entfernt[1]. In Listing 4.3 sind die vom „Java-TX Compiler“ inferierten Typen für diese Klasse zu sehen. Interessant ist vor allem der Typ des Parameters `funNParams` der Methode `createGenerics`. Dieser wurde im Vergleich zu Listing 4.1 von `List<GenericRefType>` zu `Iterable<? extends GenericRefType>`. Wie man in Abbildung 4.1 sehen kann, ist `Iterable` ein Interface, dass von `List` erweitert wird und somit ein Supertyp von List.

Die Wildcard `? extends` ermöglicht Kovarianz und somit auch Subtypen von `GenericRefType` als generischen Typparameter. Mit dem neuen Typ ist der Definitionsbereich der Methode somit offensichtlich größer als zuvor und der Typ damit genereller. Aber ist dieser Typ in diesem Kontext auch korrekt?

[1] Dies wäre auch in Java mit dem `var` Platzhalter möglich

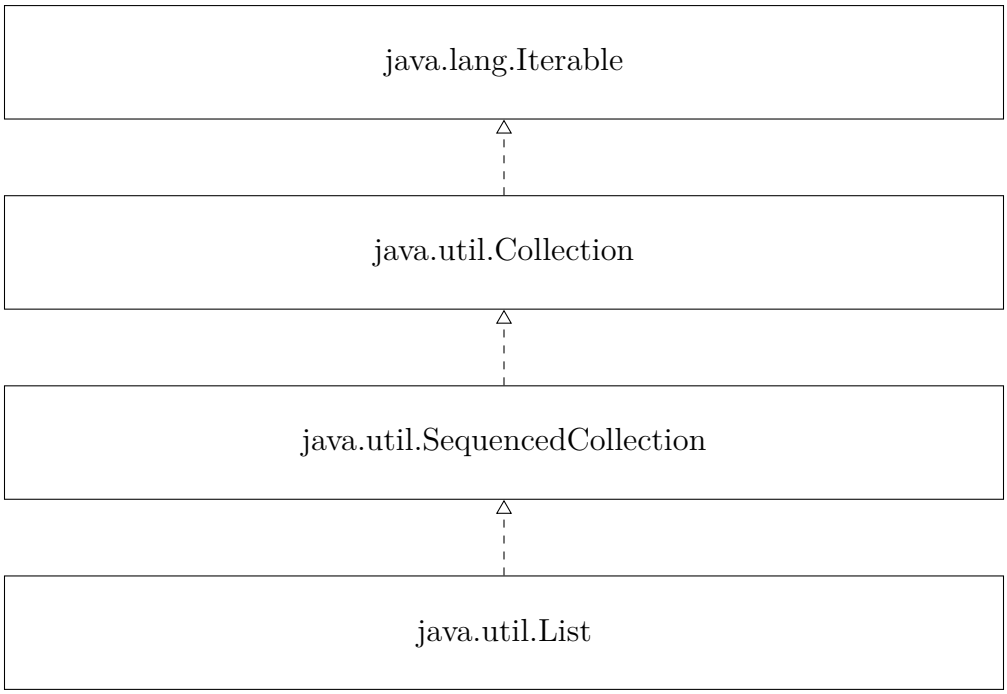

Abbildung 4.1: Vererbungshierarchie von java.util.List ab Java 21 [13]

```
1  public class FunNClass extends ClassOrInterface {
2      private static GenericDeclarationList createGenerics(Iterable
           <? extends GenericRefType>funNParams) {
3          var generics = new ArrayList<GenericTypeVar>();
4          for (GenericRefType param : funNParams) {
5              generics.add(new GenericTypeVar(param.getParsedName()
                   ,
6                       new ArrayList<>(), new NullToken(), new
                           NullToken()));
7          }
8          return new GenericDeclarationList(generics, new NullToken
               ());
9      }
10     ...
```

Listing 4.3: Inferierte Typen für Listing 4.2

Zunächst einmal wird in der Funktion die Liste lediglich durchlaufen. Es werden also keine speziellen Methoden des Interface `List` verwendet. Die For-Each Schleife ist auf alle Typen anwendbar, die das Iterable Interface implementieren. Iterable ist also der allgemeinste Typ, auf den eine For-Each Schleife angewandt werden kann [33]. Da innerhalb der Methode nur lesend auf `funNParams` zugegriffen wird, ist auch die Kovarianz des Typs unproblematisch und korrekt [17][S.19 ff]. Der Typ ist also ein korrekter Typ für den Parameter `funNParams`. Die meisten Java Programmierer hätten diesen Typen vermutlich nicht gefunden. Dies

zeigt die Stärke des Typinferenzalgorithmus von Java-TX. Dass der Rückgabewert bereits der speziellste Typ ist, ist leicht zu sehen, da in der letzten Zeile explizit ein Objekt des Typs `GenericDeclarationList` zurückgegeben wird. Alle weiteren Typen, die inferiert wurden, entsprechen den Typen in Listing 4.1 und sind somit ebenfalls korrekt.

Grundsätzlich wird der Code durch das entfernen von Typen kompakter. Es lässt sich allerdings darüber streiten, ob der Code durch das weglassen der Typinformationen auch lesbarer wird. Es kann sicherlich auch von Vorteil sein die Parameter- und Rückgabetypen einer Methode direkt zu sehen. Hier wäre das entwickelte Eclipse Plugin ein guter Kompromiss, welches den gewünschten Typ direkt in den Quellcode einfügen kann [28]. Dies ist vor allem auch deshalb sinnvoll weil es mehrere korrekte Typen geben kann [25][Abschnitt 3.1]. In diesem Fall könnte der Entwickler den gewünschten Typen auswählen.

5 Fazit und Ausblick

5.1 Fazit

Im Laufe dieser Studienarbeit konnte bisher nur ein Bruchteil des Quellcodes übersetzt werden. Abbildung 5.1 zeigt das aktuelle Verhältnis von Java zu Java-TX Quelldateien. Bisher konnten nur 18 von 251 Quelldateien erfolgreich übersetzt werden. “Erfolgreich„ bedeutet in diesem Kontext, dass alle Tests der Testsuite erfolgreich durchlaufen wurden.

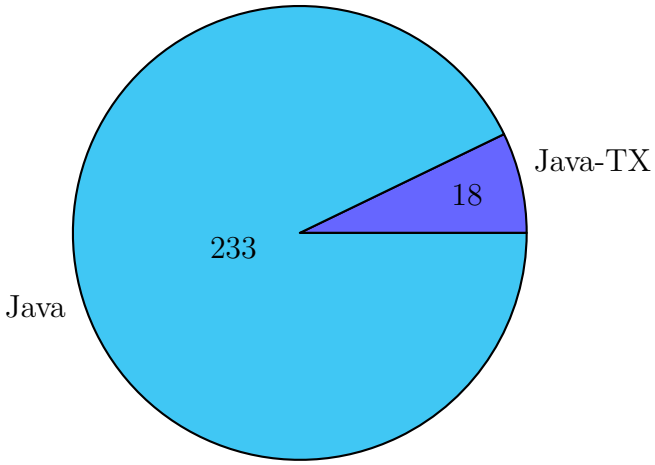

Abbildung 5.1: Verhältnis der Java und Java-TX Dateien im „Java-TX Compiler in Java-TX“

Die übersetzten Dateien beschränken sich aktuell auf die Pakete `de.dhbwstuttgart.typeinference` und `de.dhbwstuttgart.syntaxtree`. Der Umfang der übersetzten Dateien ist vergleichsweise gering. Dennoch konnten durch die Studienarbeit viele Bugs im Compiler gefunden und behoben werden, wodurch sich die generelle Qualität des Compilers verbessert hat. Einige neue Features, welche zum Übersetzen der Quelldateien in Java-TX notwendig waren, wurden ebenfalls hinzugefügt. Die größte Schwachstelle des Compilers sind aktuell wohl die Fehlermeldungen. Diese sind oft schwer zu verstehen und einzugrenzen. Hier besteht definitiv noch Verbesserungspotential. Außerdem fehlen auch aktuell noch einige der grundlegenden Funktionen von Java, wie z.B. die Main Methode, die wegen der fehlenden Unterstützung für Arrays noch nicht implementiert wurde.

5.2 Ausblick

Das langfristige Ziel wird sicherlich sein (ggf. in nachfolgenden Studienarbeiten), den gesamten Quellcode des „Java-TX Compiler“ in Java-TX zu übersetzen[1], um die Qualität und Funktionsumfang des Compilers sicherzustellen. Der Weg dahin ist jedoch noch weit. Es gibt sicherlich noch viele unentdeckte Bugs und Probleme, die es zu lösen gilt. Zudem müsste man an bestimmten Stellen nicht nur die Typinformationen entfernen, sondern den Code anpassen. Der „Java-TX Compiler“ verwendet historisch bedingt teilweise noch alte Java-Features, die in Java-TX nicht mehr unterstützt werden, z.B. Raw Types [26]. Außerdem müsste man zum aktuellen Stand des Compilers sämtliche Arrays in Listen umwandeln, da Java-TX aktuell keine Arrays unterstützt. Dies ist darauf zurückzuführen, dass Arrays auch in Java eine gewisse Sonderstellung haben, da sie aus einer Zeit vor generischen Typen stammen und im Vergleich zu Collections einige Nachteile haben [17][Abschnitt 2.5, 6.9]. Doch auch wenn Arrays im Quellcode durch Listen ersetzt werden können, gibt es gewisse Methoden in Java Bibliotheken, die Arrays verwenden. So z.B. auch die Methode `split` der Klasse `java.lang.String`, die ein Array von Strings zurückgibt [31]:

```
public final class String implements ... {
    ...
    public String[] split(String regex) {...}
    public String[] split(String regex, int limit) {...}
    ...
}
```

Diese Methoden sind z.B. aus Java-TX nicht aufrufbar, da sie Arrays zurückgeben. Langfristig wird es also notwendig sein, Arrays in Java-TX zu unterstützen, um die Kompatibilität mit Java Klassen zu gewährleisten. Arrays sind in Java auch essenziell für die Main Funktion, die daher in Java-TX auch noch fehlt.

Erstrebenswert wäre es auch, die Fehlermeldungen des Compilers weiter zu verbessern. Dies würde sicherlich auch die Fehlersuche beim Übersetzen der Quelldateien erleichtern. Auch eine vollständige Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces wäre wünschenswert, um den Vorteil der echten Funktionstypen vollumfänglich mit bestehenden Java Bibliotheken ausnutzen zu können.

[1] Externe Tools wie Antlr oder ASM, die zur Implementierung des Compilers verwendet wurden, werden natürlich weiterhin Java Code verwenden

Literatur

[1] Baeldung. *Class Loaders in Java | Baeldung*. 11. Mai 2024. URL: https://www.baeldung.com/java-classloaders (besucht am 25. 05. 2024).

[2] Baeldung. *The Java 8 Stream API Tutorial | Baeldung*. 15. Juni 2016. URL: https://www.baeldung.com/java-8-streams (besucht am 25. 05. 2024).

[3] Alex Buckley. *The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 14*. JSR 14: Add Generic Types To The JavaTM Programming Language. 30. Sep. 2004. URL: https://www.jcp.org/en/jsr/detail?id=14 (besucht am 25. 04. 2024).

[4] Maurizio Cimadamore. *JEP 101: Generalized Target-Type Inference*. JEP 101: Generalized Target-Type Inference. 22. Feb. 2011. URL: https://openjdk.org/jeps/101 (besucht am 25. 04. 2024).

[5] Joe Darcy. *The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 334*. JSR 334: Small Enhancements to the JavaTM Programming Language. 16. Nov. 2010. URL: https://jcp.org/en/jsr/detail?id=334 (besucht am 25. 04. 2024).

[6] Joseph D. Darcy. *JEP 126: Lambda Expressions & Virtual Extension Methods*. JEP 126: Lambda Expressions & Virtual Extension Methods. 1. Nov. 2011. URL: https://openjdk.org/jeps/126.

[7] *gcc-in-cxx - GCC Wiki*. URL: https://gcc.gnu.org/wiki/gcc-in-cxx (besucht am 01. 06. 2024).

[8] Brian Goetz. *JEP 286: Local-Variable Type Inference*. JEP 286: Local-Variable Type Inference. 8. März 2016. URL: https://openjdk.org/jeps/286 (besucht am 25. 04. 2024).

[9] Brian Goetz. *The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 335*. JSR 335: Lambda Expressions for the JavaTM Programming Language. 16. Nov. 2010. URL: https://www.jcp.org/en/jsr/detail?id=335 (besucht am 06. 05. 2024).

[10] James Gosling u. a. *The Java Language Specification, 3rd Edition*. 3. Aufl. Upper Saddle River, NJ: Addison Wesley, 14. Juni 2005. 684 S. ISBN: 978-0-321-24678-3.

[11] James Gosling u. a. *The Java® language specification*. Java SE 8 edition. OCLC: ocn873760233. Upper Saddle River, NJ: Addison-Wesley, 2014. 758 S. ISBN: 978-0-13-390069-9.

[12] Atsushi Igarashi und Mirko Viroli. „On Variance-Based Subtyping for Parametric Types“. In: *ECOOP 2002 — Object-Oriented Programming*. Hrsg. von Boris Magnusson. Bearb. von Gerhard Goos, Juris Hartmanis und Jan Van Leeuwen. Bd. 2374. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 441–469. ISBN: 978-3-540-43759-8 978-3-540-47993-2. DOI: 10.1007/3-540-47993-7_19. URL: http://link.springer.com/10.1007/3-540-47993-7_19 (besucht am 01. 06. 2024).

[13] *Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html (besucht am 29. 05. 2024).

[14] Jetbrains. *Java Programming - The State of Developer Ecosystem in 2023 Infographic*. JetBrains: Developer Tools for Professionals and Teams. 2023. URL: https://www.jetbrains.com/lp/devecosystem-2023 (besucht am 01. 06. 2024).

[15] Simon Marlow u. a. „Haskell 2010 language report“. In: *Available online http://www.haskell. org/(May 2011)* (2010).

[16] *Maven – Introduction*. URL: https://maven.apache.org/what-is-maven.html (besucht am 20. 05. 2024).

[17] Maurice Naftalin und Philip Wadler. *Java generics and collections*. OCLC: ocm76810468. Beijing ; Sebastopol, CA: O'Reilly, 2007. 273 S. ISBN: 978-0-596-52775-4.

[18] Rafael del Nero. *All about Java class loaders*. InfoWorld. 29. Juni 2023. URL: https://www.infoworld.com/article/3700054/all-about-java-class-loaders.html (besucht am 25. 05. 2024).

[19] *OpenJDK Repository - Javac*. GitHub. URL: https://github.com/openjdk/jdk/blob/master/src/jdk.compiler/share/classes/com/sun/tools/javac (besucht am 01. 06. 2024).

[20] *OpenJDK: Project Lambda*. URL: https://openjdk.org/projects/lambda/ (besucht am 06. 05. 2024).

[21] Oracle. *Type Inference (The Java™ Tutorials > Learning the Java Language > Generics (Updated))*. URL: https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html (besucht am 01. 06. 2024).

[22] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Massachusetts London, England: The MIT Press, 2002. 623 S. ISBN: 978-0-262-16209-8.

[23] Adrien Piquerez, Jamie Thompson und et. al. *Variances*. Scala Documentation. URL: https://docs.scala-lang.org/tour/variances.html (besucht am 30. 05. 2024).

[24] Martin Plümicke und Andreas Stadelmeier. „Introducing Scala-like function types into Java-TX“. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. New York, NY, USA: Association for Computing Machinery, 27. Sep. 2017, S. 23–34. ISBN: 978-1-4503-5340-3. DOI: 10.1145/3132190.3132203. URL: https://doi.org/10.1145/3132190.3132203 (besucht am 24. 04. 2024).

[25] Martin Plümicke und Etienne Zink. *Java-TX: The language*. Fakultät Technik der Dualen Hochschule Baden-Württemberg Stuttgart, Jan. 2022. URL: https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights/.

[26] *Raw Types (The Java™ Tutorials > Learning the Java Language > Generics (Updated))*. URL: https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html (besucht am 02. 06. 2024).

[27] SoftwareAlchemy. *Streamline Your Java Code: A Cheat Sheet for Mastering Streams*. Javarevisited. 23. Apr. 2024. URL: https://medium.com/javarevisited/streamline-your-java-code-a-cheat-sheet-for-mastering-streams-e8500f4495fe (besucht am 25. 05. 2024).

[28] Andreas Stadelmeier. „Java type inference as an Eclipse plugin“. In: 45. Jahrestagung der Gesellschaft für Informatik, Informatik 2015, Informatik, Energie und Umwelt. Cottbus, Deutschland, 28. Sep. 2015, S. 1841–1852. URL: https://subs.emis.de/LNI/Proceedings/Proceedings246/article18.html.

[29] Richard Stallman und GCC Developer Community. *Using the GNU Compiler Collection For gcc version 14.1.0*. 2024. URL: https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc.pdf.

[30] Richard Stallman, Roland McGrath und Paul D. Smith. *GNU Make: a program for directing recompliation ; GNU make version 3.81*. Boston, Mass: Free Software Foundation, 2004. 184 S. ISBN: 978-1-882114-83-2.

[31] *String (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html (besucht am 02. 06. 2024).

[32] Patrick D. Terry. *Compilers and compiler generators: an introduction with C++*. London: International Thomson Computer Press, 1997. 579 S. ISBN: 978-1-85032-298-6.

[33] *The For-Each Loop*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html (besucht am 23. 05. 2024).

[34] Simon Thompson. *Haskell: the craft of functional programming*. 3rd ed. Harlow, England ; New York: Addison Wesley, 2011. 585 S. ISBN: 978-0-201-88295-7.

[35] David Vandevoorde und Nicolai M. Josuttis. *C++ templates: the complete guide*. Boston, Mass.: Addison-Wesley, 2010. 528 S. ISBN: 978-0-201-73484-2.

[36] Bill Wagner u. a. *Creating Variant Generic Interfaces - C#*. 15. Sep. 2021. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/creating-variant-generic-interfaces (besucht am 30. 05. 2024).

Anhang

.1 Sourcecode des Bash Skripts zur Kompilierung

```
#!/bin/bash

SRCDIR="javatx-src/main/java"
DESTDIR="out/src"
TESTDESTDIR="out/tests"
DEPENDENCIES="dependencies/*"
JAVAC_FLAGS="-g:none -nowarn"
JAVATX_COMPILER_PATH="JavaTXcompiler.jar"
COMPILED_CLASSES="lib/classes"

#remove all files, if the script is called with parameter "clean"
if [ "$1" = "clean" ]; then
    rm -r "$DESTDIR"
    exit 0
fi

if [ "$1" != "" ]; then
  echo "invalid argument: $1"
  exit 1
fi

#find all .java/.jav files and store paths in an array
#note: somehow absolute paths don't work correctly with find -newer
#JAVA_FILES=($(find "$SRCDIR" -name "*.java" -exec realpath {} \;))
#JAV_FILES=($(find "$SRCDIR" -name "*.jav" -exec realpath {} \;))
JAVA_FILES=($(find "$SRCDIR" -name "*.java"))
JAV_FILES=($(find "$SRCDIR" -name "*.jav"))


#create empty arrays for .class file paths
JAVA_CLASSES=()
```

```
32 JAV_CLASSES=()
33
34 JAVA_CHANGED=()
35 JAV_CHANGED=()
36
37 mkdir -p $DESTDIR
38
39 #fill class files arrays by subsituting .java/.jav -> .class for
       each file
40 for file in "${JAVA_FILES[@]}"; do
41     #substitute destination dir with source dir
42     class_name="$DESTDIR${file#$SRCDIR}"
43     #substitute *.java -> *.class
44     class_name="${class_name%.java}.class"
45     #if .class file does not exists or .class file older than .
           java file
46     if [ ! -f "$class_name" ] || [ "$(find "$file" -prune -newer
           "$class_name")" ]; then
47         JAVA_CHANGED+=("$file")
48         JAVA_CLASSES+=("$class_name")
49     fi
50 done
51
52
53 for file in "${JAV_FILES[@]}"; do
54     #substitute destination dir with source dir
55     class_name="$DESTDIR${file#$SRCDIR}"
56     #substitute *.jav -> *.class
57     class_name="${class_name%.jav}.class"
58     #if .class file does not exists or .class file older than .
           jav file
59     if [ ! -f "$class_name" ] || [ "$(find "$file" -prune -newer
           "$class_name")" ]; then
60         JAV_CHANGED+=("$file")
61         JAV_CLASSES+=("$class_name")
62     fi
63 done
64
65
66 if [ "${#JAV_CHANGED[@]}" -ne 0 ]; then
```

```
67      echo "java -jar $JAVATX_COMPILER_PATH -d $DESTDIR -cp "
            $SRCDIR:$DEPENDENCIES:$COMPILED_CLASSES" ${JAV_CHANGED[@]}"
68      java -jar $JAVATX_COMPILER_PATH -d $DESTDIR -cp "$SRCDIR:
            $DEPENDENCIES:$COMPILED_CLASSES" "${JAV_CHANGED[@]}"
69      if [ $? -ne 0 ]; then
70          echo "Fehler beim Kompilieren der Jav-Dateien. Beende das
                Skript."
71          exit 1
72      fi
73  fi
74
75  if [ "${#JAVA_CHANGED[@]}" -ne 0 ]; then
76      echo "javac -d $DESTDIR -cp "$SRCDIR:$DESTDIR:$DEPENDENCIES"
            $JAVAC_FLAGS ${JAVA_CHANGED[@]}"
77      javac -d $DESTDIR -cp "$SRCDIR:$DESTDIR:$DEPENDENCIES"
            $JAVAC_FLAGS "${JAVA_CHANGED[@]}"
78  fi
```