Bachelorarbeit
Bytecode-Generierer für ein typploses Java

Daniel Holle

29. Oktober 2022

Betreuer	Prof. Dr. Martin Plümicke
	Andreas Stadelmeier
Matrikelnummer	4169714
Studiengang	Medieninformatik BSc 2015

1 Abstract

JavaTX ist eine Programmiersprache basierend auf Java 8, welche die bekannte Syntax durch eine globale Typinferenz erweitert. Damit können Typen im gesamten Programm durch den Compiler inferiert werden. Im Rahmen dieser Arbeit soll der Codegenerierer überarbeitet werden. Ziel ist es, durch neue Java Features eine einfachere und saubere Implementierung zu erreichen. Ebenfalls sollen die durch JavaTX inferierten Typen als generierte Generics im kompilierten Bytecode eingeführt werden. Es wird ein Überblick zu den verwendeten Konzepten geschaffen. Im Anschluss wird das Visitor-Pattern mit dem neu in Java 18 eingeführten Typswitch verglichen und überprüft, ob das verwendete Konzept gut umgesetzt wurde.

Inhaltsverzeichnis

2 Einleitung

2.1 Java und JavaTX

Die Programmiersprache Java hat sich seit ihrer Entstehung stetig weiterentwickelt. Mit Java 5 wurden Generics hinzugefügt, ein Konzept, das in funktionalen Programmiersprachen als parametrische Polymorphie bekannt ist. Damit ist es möglich, Klassen zu erstellen, die über die verwendeten Typen abstrahieren. Seit Java 8 gibt es Lambda Ausdrücke, die auf Functional Interfaces basieren. Ein solches Interface besteht im Wesentlichen aus einer abstrakten Funktion, die dann vom Lambda Ausdruck implementiert wird. Die Vor- und Nachteile dieses Ansatzes werden in [6] diskutiert. Seit Java 9 gibt es eine limitierte Form der Typinferenz die es erlaubt, den Typ von lokalen Variablen wegzulassen. Dieser wird dann vom Compiler inferiert.

JavaTX (steht für Java Type eXtended) ist eine auf Java 8 basierende Programmiersprache, die Java um globale Typinferenz, wie sie aus funktionalen Sprachen wie Haskell bekannt ist, erweitert. Damit ist es möglich, Typen auch in anderem Kontext inferieren zu lassen, wie bei Feldern und Funktionen.

Weiterhin wird Java durch echte Funktionstypen erweitert, wie sie beispielsweise in Scala zu finden sind. Der Vorteil dabei besteht darin, dass der Typ von Lambda Ausdrücken inferiert werden kann und nicht wie bei Java aus dem Kontext oder explizit definiert wird. Als Drittes werden Generics für Klassen und Funktionen generiert, mit denen freie Typvariablen generalisiert werden [8].

2.2 Motivation

Der Codegenerierer von JavaTX in der jetzigen Form ist zwar funktional, aber die Codequalität lässt zu wünschen übrig. Es gibt viele Stellen, an denen globaler Zustand verwendet wird, was aus Gründen der geringen Lesbarkeit des Codes und die schlechte Wartbarkeit zu vermeiden ist. Auch wird gerade durch das Visitorpattern sehr viel Code benötigt, der durch die neuen Java-Features rund um Switch-Statements stark vereinfacht werden kann. Andere Features sorgen ebenfalls dafür, dass entweder weniger Code benötigt wird oder aber eine bessere Implementierung benutzt werden kann.

Des weiteren wird durch die direkte Verwendung des ASTs des Parsers eine starke Kupplung eingeführt, die durch die Verwendung eines seperaten ASTs verringert werden kann. Der Vorteil hierbei ist, dass Änderungen am Parser und der dazugehörigen Infrastruktur nur eine Anpassung der Übersetzungsebene nötig machen, der Codegenerierer aber bestenfalls nicht verändert werden muss.

Neu eingeführt wird die automatische Generierung der Java-Generics aus freien Typvariablen. So können komplexe Relationen von Typen in den generierten Klassen abgebildet werden. Ein großer Vorteil ist die einfache Interoperabilität zu klassischem Java-Code. Hierbei wird möglichst der Prinzipaltyp (siehe 3.1.2) durch Java-Generics abgebildet, der klassischerweise nur durch den Objekttyp erreicht werden kann. Es gibt also eine deutlich feinere Abstufung von Typen, die jeweils durch eigene Typschranken genauer spezifiziert sind.

Der Algorithmus zur Generierung von Generics wird in [8] vorgestellt, in dieser Arbeit soll lediglich eine Implementierung geschrieben werden. Ein Ziel hierbei ist, dass der Code möglichst eigenständig funktioniert. Man soll also eine Komponente innerhalb des Kompilers bekommen, die modular abgegrenzt ist.

Zuletzt soll auch eine Überladung von Funktionstypen erreicht werden. Dazu werden für sämtliche Typisierungen einer Funktion eine seperate Klasse

generiert, damit diese auch nach Java ein seperater Typ sind.

3 Grundlagen

3.1 Typinferenz

3.1.1 Was ist Typinferenz?

In Java hat jeder Ausdruck, wie bei anderen statischen Programmiersprachen, einen konkreten Typ. Der Unterschied zu einer Sprache mit Typinferenz besteht darin, dass diese Typen aus dem Kontext ermittelt werden. Hierbei ist allerdings festzustellen, dass anders als bei dynamischen Programmiersprachen wie Python, weiterhin jeder Ausdruck einen statischen Typ hat. Es ist also nicht möglich, einer Variablen Ausdrücke mit verschiedenen Typen zuzuweisen.

In Java gibt es bereits seit Version 7 eine einfache Form der Typinferenz, den sogenannten Diamond-Operator:

```
1 // Java 6
2 ArrayList<String> myList = new ArrayList<String>();
3 // Java 7
4 ArrayList<String> myList = new ArrayList<>();
```

Bei dieser einfachen Form der Typinferenz kann man bereits die Vorteile der Typinferenz gut abschätzen. Es ist nicht mehr nötig, den Typ `String` mehrfach anzugeben. Das spart Tipparbeit und damit ist der Code auch weniger anfällig für Fehler. Allerdings muss man bei diesem Beispiel immer noch den Typ `ArrayList` mehrfach angeben.

Seit Java 9 gibt es hierfür eine Form der Typinferenz für lokale Variablen, mit dem neuen Schlüsselwort `var`:

```
1 // Java 9
2 var myList = new ArrayList<String>();
```

Diese Form der Typinferenz ist trivial, da der Compiler bereits für das Typchecking den Typ des Ausdrucks auf der rechten Seite der Zuweisung berechnen muss. Für die Typinferenz wird also lediglich dieser Typ für die Variable inferiert. Ohnehin sind Typchecking und Typinferenz sehr stark miteinander verwandt. Zuerst wird die Typinferenz angewendet und dann mit dem Typchecker überprüft, ob die Typen konsistent sind.

Die Typinferenz in Java ist allerdings nur sehr eingeschränkt möglich. So muss beispielsweise für die Typinferenz einer lokalen Veriablen direkt eine Zuweisung erfolgen:

```
1 // Invalid
2 var myList;
3 myList = new ArrayList<String>();
```

Obwohl hier der Typ von `myList` eindeutig aus dem Kontext klar wird, kann Java diesen Typ nicht inferieren.

Weiterhin ist es nicht möglich, Typen in anderem Kontext zu inferieren. Beispielsweise müssen für Funktionen immer alle Parameter typisiert sein sowie der Rückgabetyp. Auch ist es nicht möglich, den Typ von Instanzvariablen zu berechnen.

In JavaTX werden diese Restriktionen aufgehoben und es ist möglich, Typen in sämtlichen Kontexten zu inferieren. Der Algorithmus hierfür wird in [7] vorgestellt. Er basiert auf dem Algorithmus W von Milner und Damas [3], der durch Subtypisierung von Java Klassen ergänzt wurde. Zusammengefasst wird ein Set von Constraints erstellt, welches durch die eingesetzten Typen erfüllt wird. Hierbei kann es mehrere Lösungen geben, wofür Java passenderweise das Overloading von Methoden erlaubt. Das heißt, eine Methode kann

mehrmals angegeben werden, so lange die Parameter verschiedene Typen haben. JavaTX generiert also teils mehrere Methoden für eine Definition, was einiges an Arbeit erspart.

Gegeben sei folgendes JavaTX Programm:

```
1  import java.lang.Integer;
2  class Main {
3      x = 42;
4      f (y, z) {
5          if (y) {
6              return z + x;
7          } else {
8              return 0;
9          }
10     }
11 }
```

Das Resultat ist die folgende dekompilierte Klasse:

```
1  public class Main {
2      public Integer x = 42;
3
4      public Main() {
5      }
6
7      public Integer f(Boolean y, Integer z) {
8          return y ? z + this.x : 0;
9      }
10 }
```

Man sieht, dass für `y` der Typ Boolean ausgewählt wurde, da der Typ in einem If-Statement verwendet wird, welches einen Boolean akzeptiert. Für `x` wurde einfach der Typ des Integer Literals verwendet, ähnlich wie zuvor bei der Zuweisung von einer lokalen Variable. Für `z` wurde der Typ Integer ausgewählt, da bei der Addition zweier Integer ein weiterer Integer heraus kommt. Hier hätten theoretisch auch andere Typen eingesetzt werden können, aber JavaTX betrachtet aus Performancegründen nur die importierten

Datentypen. Wenn nun statt `java.lang.Integer java.lang.Double` importiert wird, sieht die Klasse folgendermaßen aus:

```
1  public class Main {
2      public Double x = (double)42;
3
4      public Main() {
5      }
6
7      public Double f(Boolean y, Double z) {
8          return y ? z + this.x : (double)0;
9      }
10 }
```

Man sieht also, dass für x und z jeweils verschiedene Typen eingesetzt werden können.

Bei diesen einfachen Beispielen wird noch nicht ganz klar, wie mächtig Typinferenz sein kann. Gerade bei komplizierten generischen Typen mit mehreren Typparametern und Wildcard-Typen wird viel Code eingespart. Das liegt teilweise auch daran, dass Java keine Möglichkeit bietet, einen Typalias zu erstellen.

Im Stream Interface der Java Standardbibliothek werden mehrere Methoden mit komplizierten Signaturen definiert, wie zum Beispiel `reduce`:

```
1  public interface Stream<T> extends BaseStream<T, Stream<T>> {
2      <U> U reduce (
3          U identity,
4          BiFunction<U, ? super T, U> accumulator,
5          BinaryOperator<U> combiner
6      );
7      ...
8  }
```

3.1.2 Prinzipaltyp

In [10] wird die Prinzipaltyp-Eigenschaft vorgestellt:

> Gegeben: Ein typisierbarer Term M
>
> Es existiert: Eine Typisierung $A \vdash M : \sigma$
>
> repräsentiert alle möglichen Typisierungen von M

Es ist sinnvoll, für ein Typssystem diese Eigenschaft zu besitzen, da es die Möglichkeit eröffnet für einen beliebigen Ausdruck einen Typ herzuleiten, welcher alle möglichen Typisierungen einschließt. Ein Typinferenzalgorithmus berechnet genau diesen Prinzipaltyp.

Für Java wird diese Definition nach [8] angepasst, damit die Prinzipaltypen dem maximalen Element der Subtyp-Relation entsprechen:

> „Ein Schnitttyp mit der minimalen Anzahl an Elementen für eine Deklaration entspricht dem Prinzipaltyp, wenn jedes andere Typschema für die Deklaration ein generischer Subtyp eines Elements des Schnitttyps ist.“

3.2 Sealed Classes

In Java werden typischerweise Vererbungshierarchien verwendet, um verschiedene Aktionen, basierend auf unterschiedlichen Datentypen, auszuführen:

```
1 public interface Animal {
2     public void talk();
3 }
4 public class Dog implements Animal {
5     @Override
6     public void talk() {
7         System.out.println("wuff");
8     }
```

```
9  }
10 public class Cat implements Animal {
11     @Override
12     public void talk() {
13         System.out.println("meow");
14     }
15 }
```

Hier wird beispielsweise die Methode `talk` überschrieben, was dazu führt, dass bei dem Aufruf von `talk` auf dem Interface `Animal` jeweils die korrekte Methode ausgewählt wird, je nachdem um welchen Typen es sich handelt. Ein großer Vorteil dieser Herangehensweise ist, dass `Animal` beispielsweise außerhalb einer Bibliothek vom Benutzer der Bibliothek erweitert werden kann.

Manchmal ist es jedoch nicht erwünscht, dass der Nutzer der Bibliothek die Klassen beliebig erweitern kann. Das kann etwa dann der Fall sein, wenn die Klasse nicht für Vererbung vorgesehen wurde und es Code gibt, welcher nur für bestimmte Subklassen funktioniert. In früheren Versionen von Java konnte dies nur beispielsweise durch das Schlüsselwort `final` erreicht werden, welches allerdings die Klasse komplett für Vererbung ausschließt.

Je nach Modellierung des Problems kann es sinnvoll sein, die Logik für verschiedene Fälle je nach Typ einer Instanz in einer Funktion zu implementieren, anstatt auf Vererbung zu setzen.

Ein einfaches Beispiel einer solchen Implementierung kann auf Enums basieren:

```
1 enum Animal { CAT, DOG }
2
3 static void talk(Animal animal) {
4     switch (animal) {
5     case CAT:
6         System.out.println("meow");
```

```
7           break;
8       case DOG:
9           System.out.println("wuff");
10          break;
11      }
12  }
```

Das Problem bei solch einer Implementierung ist, dass `CAT` und `DOG` jeweils ein Singleton sind und deshalb keine Instanzvariablen besitzen können. Es wäre also sinnvoll, einen Switch über den Typ einer Objektinstanz zu benutzen. In älternen Versionen von Java sähe das in etwas so aus:

```
1  public interface Animal {}
2
3  static void talk(Animal animal) {
4      if (animal instanceof Cat) {
5          Cat cat = (Cat) animal;
6          ...
7      } else if (animal instanceof Dog) {
8          ...
9      }
10 }
```

Das Problem hierbei ist, dass Animal von außerhalb erweitert werden kann und `talk` dann nicht das korrekte Ergebnis liefern kann. Ebenso wird nicht zur Kompilierzeit überprüft, ob alle Fälle abgedeckt werden.

In [1] wird, um dieses Problem zu beheben, ein neuer Typ von Klasse eingeführt, nämlich die sogenannten Sealed-Klassen. Mit dem modifier `sealed` können nur bestimmte Subklassen die Klasse erweitern. Die Syntax dafür sieht so aus:

```java
public sealed interface Animal permits Cat, Dog  {}

public final class Cat implements Animal {}
public final class Dog implements Animal {}

static void talk(Animal animal) {
    switch (animal) {
        case Dog dog:
            System.out.println("wuff");
            break;
        case Cat cat:
            System.out.println("meow");
            break;
    }
}
```

Man sieht also, dass das Interface `Animal` nur die Subtypen `Cat` und `Dog` zulässt. Diese beiden Klassen müssen mit `final` markiert werden, damit sie an anderer Stelle nicht erweitert werden können und damit die Einschränkung des Interfaces `Animal` übergehen. Es ist allerdings möglich, durch ein weiteres Schlüsselwort `non-sealed` diese Einschränkung zu übergehen. Da dies allerdings nur durch Klassen möglich ist, die ohnehin in der Permits-Klausel erwähnt werden müssen, bleibt die Kontrolle darüber bei der implementierung der Bibliothek, in der das Sealed Interface verwendet wird. Weiterhin ist es möglich, die Klassen ebenfalls mit `sealed` zu markieren und eine eigene Permits-Klausel zu erstellen.

Ebenfalls zu erwähnen, ist diese neue Form des Switch-Statements. Es wird, wie bei dem Beispiel davor, über den Typ des Ausdrucks `animal` geswitched. Diese Form des Switch-Statements wird in der Implementierung des Codegenerieres noch öfters verwendet, auf die Details dazu wird in 5.1.2 eingegangen.

3.3 Recordtypen

In Java 15 und [5] wird ein neues Schlüsselwort für die Erstellung von Klassen hinzugefügt: `record`. Java kann an manchen Stellen sehr wortreich sein, gerade wenn es um die Erstellung von POD-Typen, das heißt Klassen, die als unveränderbare Daten-Container gedacht sind, geht. Es müssen hier mehrere Methoden implementiert werden, die alle nach Schema-F aufgebaut sind. Hierzu gehören die Getter-Methoden für die unveränderbaren Datenfelder, die alle `final` sind und im Konstruktor übergeben werden. Ebenfalls implementiert werden müssen die Methoden `hashCode` und `equals`, die jeweils nötig sind, um Objekte der Klasse in einer `HashMap` zu verwenden. Des Weiteren kann auch noch die Methode `toString` implementiert werden, die eine textuelle Ausgabe des Objekts ermöglicht.

Eine Klasse, die alle diese Methoden implementiert, sieht demnach so aus:

```
1  public class Employee {
2      private final String firstName;
3      private final String lastName;
4
5      public Employee(String firstName, String lastName) {
6          this.firstName = firstName;
7          this.lastName = lastName;
8      }
9
10     public String firstName() { return firstName; }
11     public String lastName() { return lastName; }
12
13     public boolean equals(Object o) {
14         if (!(o instanceof Employee)) return false;
15         Employee other = (Employee) o;
16         return other.firstName.equals(this.firstName) &&
17             other.lastName.equals(this.lastName);
18     }
19     public int hashCode() {
```

```
20          return Objects.hash(firstName, lastName);
21      }
22      public String toString() {
23          return "Employee { " + firstName + " " + lastName + "
                }";
24      }
25  }
```

Man sieht, dass bei Implementierung der `equals` Methode schnell Fehler passieren können. Ebenso wird sehr viel Code dupliziert, wenn es mehrere Klassen gibt, die in dieses Schema passen.

Andere Sprachen, die auf der JVM basieren, wie beispielsweise Kotlin, haben bereits ein Konzept namens Daten-Klassen, die all diese Methoden automatisch generieren.

In Java 15 werden eben diese Record-Klassen vorgestellt. Die eingangs vorgestellte Klasse würde demnach wie im untenstehenden Code aussehen:

```
1  public record Employee(String firstName, String lastName) {}
```

Dabei ist zu beachten, dass eine Klasse generiert wird, die von `java.lang.Record` erbt, das heißt, anders als beispielsweise in Kotlin, kann ein Record von keiner anderen Klasse erben.

Eine weitere Einschränkung besteht darin, dass die Klasse selbst implizipt `final` ist, also ebenso keine anderen Klassen von ihr erben können. Allerdings kann ein Record Interfaces implementieren.

Durch die Verbindung von Record-Klassen mit Sealed Interfaces können mit Java algebraische Datentypen implementiert werden. Die Sealed Interfaces beschreiben demnach Summentypen und die Record-Klassen sind Produkttypen.

Gerade für die Implementierung des neuen Target-ASTs sind Record-Klassen sehr sinnvoll. Das liegt daran, dass jeweils die ganze Logik zur Traversierung

außerhalb des ASTs definiert ist, und die einzelnen Nodes demnach nur als Container für ihren Zustand definiert sind. Allerdings gibt es auch Nachteile. So kann beispielsweise `TargetConstructor` nicht von `TargetMethod` erben, was dazu führt, dass in diesem Fall mehrere Record-Typen implementiert werden, welche die selben Felder besitzen.

4 Implementierung

4.1 Konzeption

Um die Ziele zu erreichen gibt es mehrere Alternativen. Die sicherlich naheliegenste Möglichkeit wäre die Anpassung des vorhandenen Codegenerieres. Da es allerdings viel Zeit benötigen würde, diesen zu verstehen und die Codequalität nicht den neusten Standards entspricht, scheidet diese Möglichkeit aus. Es wird also eine komplett neue Implementierung des Codegenerieres benötigt.

Als nächste Alternative wäre zu betrachten, ob der schon vorhandene AST im neuen Codegenerierer verwendet werden kann. Ein großer Vorteil hierbei ist, dass es nicht zur Duplizierung von verschiedenen Kontrollstrukturen kommt. Ein If-Statement beispielsweise sieht im AST des Parsers und im AST des Codegenerieres nahezu gleich aus. Die meisten Kontrollstrukturen werden eins zu eins übersetzt. In Zukunft müssen mehrere Stellen angepasst werden, wenn ein neues Konstrukt geparst werden soll.

Die Vorteile bei einer Neuschreibung des Target-ASTs sind allerdings, dass es so möglich ist, eine geringere Kupplung des Parsers und des Codegenerieres über die Schnittstelle des ASTs zu erreichen. Wenn nun also der Parser geändert wird, muss lediglich die Übersetzung des ASTs angepasst werden. Auch kann es sein, dass später der Codegenerierer Optmierungen durchführt, für die ein Zustand im AST hinterlegt werden muss. In Zukunft divergieren also AST des Parsers und des Codegenerieres.

Ein großer Vorteil der Konzeption als komplett neues Modul des Compilers ist, dass der vorhandene Code nicht geändert werden muss. Als Teil eines

größeren Projektes macht es Sinn, Module möglichst unabhängig voneinander zu machen, damit bei Änderungen nur das Modul verstanden werden muss, nicht aber das ganze Projekt.

Für die Implementierung des Codegenerieres muss eine Traversierung des ASTs stattfinden. Klassisch wird so etwas mit dem Visistor-Pattern implementiert. Da es nun aber seit Java 18 eine neue Möglichkeit gibt, diese Traversierung mit Hilfe eines Typeswitches zu implementieren, gibt es eine Alternative zum Visistor-Pattern. In Sektion 5.1.2 werden beide Ansätze vorgestellt und miteinander verglichen.

4.2 Target-AST

Als Eingabe für den Code-Generierer wurde ein neuer abstrakter Syntaxbaum implementiert. Das Ziel hierbei war es, einen Schritt näher in Richtung Bytecode zu gehen, um eine möglichst direkte Transformation zu erlauben. Dazu wurden die Typplatzhalter, die von der Typunifikation in den Syntaxbaum geschrieben worden sind, durch konkrete Typen ersetzt. Neu eingeführt wurden hierbei die primitiven Datentypen, die nötig sind, um Funktionen aus der Java Standardbibliothek aufrufen zu können.

Die Implementierung benutzt zum großen Teil die in Java 15 eingeführten Record-Typen [5]. Dadurch wird die Vererbung innerhalb des Target-ASTs vereinfacht, da Record-Typen nur Interfaces implementieren, aber nicht von anderen Klassen erben können. Stattdessen wurden für die Implementierung Sealed Interfaces verwendet [1]. Dies macht es möglich, bei der Bytecode-Generierung einen Typswitch anstelle des Visitor-Patterns zu verwenden.

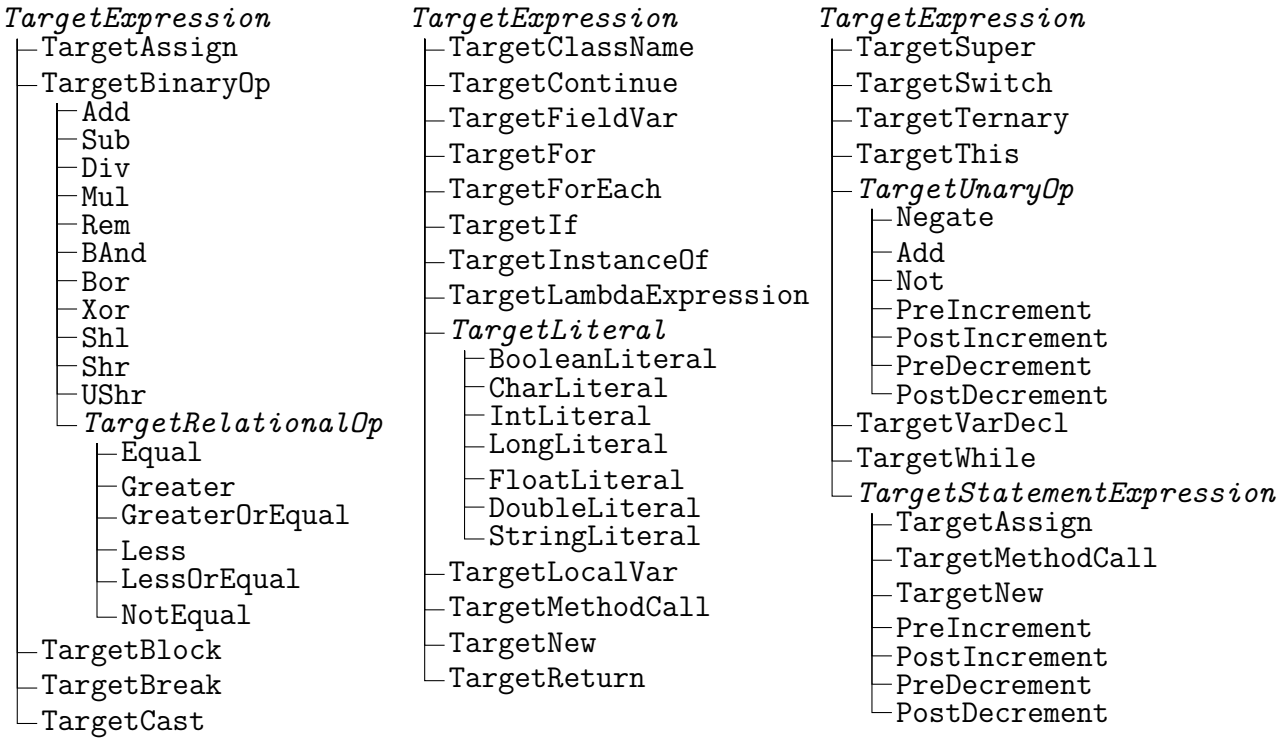

Abbildung 1: Hierarchie Target-AST

Ein weiterer Vorteil der Neuimplementierung des Target-ASTs ist, dass eine bessere Trennung des Codegenerieres und des Parsers ermöglicht wird. Wenn zuvor eine Änderung am AST gemacht wurde, musste sowohl der Parser als auch der Codegenerierer angepasst werden, um auf die neue Änderung zu reagieren. Die Trennung des Parsers vom Codegenerierer durch die Übersetzung des ASTs führt dazu, dass lediglich diese Übersetzung geändert werden muss, anstatt den Codegenerierer zu verändern.

4.3 Einsetzen des Unify-Ergebnisses

Gegeben sei ein einfaches Beispiel aus der Testsuite des Compilers, welches iterativ die Fakultät berechnet:

```
1  import java.lang.Integer;
2
3  public class Fac {
4      getFac(n) {
5          var res = 1;
6          var i = 1;
7          while (i <= n) {
8              res = res * i;
9              i++;
10         }
11         return res;
12     }
13 }
```

Listing 1: Fac.jav

Hier müssen mehrere Typen berechnet werden. Die Typinferenz arbeitet immer von Innen nach Außen, also wird zunächst der Rumpf der Methode betrachtet. Die Typen von res und i sind einfach zu finden, da sie gleich zugewiesen werden. Da `java.lang.Integer` importiert wird, bekommen beide den Typ `Integer`. Um den Typ von Parameter `n` zu berechnen, wird der Ausdruck `i <= n` betrachtet. Der Typ des Ausdrucks ist `Boolean`, da der `<=` Operator einen Boolschen Wert zurück gibt. Da `Integer` nur mit anderen Zahlen verglichen werden kann und es durch das Import-Statement eine weitere Einschränkung gibt, wertet der Typ ebenfalls zu `Integer` aus.

Das Ergebnis der Typinferenz ist ein AST mit Typvariablen und ein Set von Constraints.

```
class Fac {
    Fac(){(
        super(());
    )}::TPH AD

    TPH M getFac(TPH N n)({
        TPH O res;
        (res)::TPH O = 1;
        TPH Q i;
        (i)::TPH Q = 1;
        while((i)::TPH Q | (n)::TPH N)({
            (res)::TPH O = (res)::TPH O | (i)::TPH Q;
            (i)::TPH Q++;
        })::TPH V;

        return (res)::TPH O;
    })::TPH X

    Fac(){(
        super(());
    )}::TPH AA
}
```

Listing 2: AST von Fac.jav mit Typplatzhaltern

```
[[  (TPH T = java.lang.Integer),
    (TPH S = java.lang.Boolean),
    (TPH N = java.lang.Integer),
    (TPH M = java.lang.Integer),
    (TPH P = java.lang.Integer),
    (TPH R = java.lang.Integer),
    (TPH Q = java.lang.Integer),
    (TPH O = java.lang.Integer),
    (TPH U = java.lang.Integer)]]
```

Listing 3: Resultat der Typinferenz

In diesem Fall besteht das Resultat der Typinferenz aus einem Set. Es können aber auch mehrere Möglichkeiten existieren (Siehe 3.1.2).

Im ersten Schritt werden Constraints der Form ($T \doteq$ RefType) ausgewertet. Das bedeutet, die Typvariable T entspricht dem konkreten Typ RefType. Dafür wird eine `HashMap concreteTypes` von Typplatzhalter auf Referenztyp angelegt. Ebenfalls ausgewertet werden Constraints der Form ($T \doteq$ TypePlaceholder). Diese werden in eine seperate `HashMap equals` eingefügt.

Wenn nun ein Typplatzhalter durch einen konkreten Typ ersetzt werden soll, wird zunächst in `equals` nach dem Typplatzhalter gesucht. Wenn dieser vorhanden sein sollte, wird er durch den Typplatzhalter aus der `HashMap` ersetzt. Im zweiten Schritt wird dann in `concreteTypes` nach dem Referenztyp gesucht. Falls dieser vorhanden sein sollte, wird der konkrete Typ in einen `TargetType` umgewandelt. Falls kein konkreter Typ existiert, handelt es sich um eine freie Typvariable und sie wird durch einen generischen Typ ersetzt, der den selben Namen hat wie der Typplatzhalter.

4.4 Generierung von Generics

Wie in 4.3 beschrieben, werden zunächst konkrete Typen in den AST eingesetzt. Diese werden im Unify-Ergebnis mit ($T \doteq$ RefType) angegeben. Übrig bleiben Constraints der Form ($T \lessdot S$), wobei T und S beides Typvariablen sind. In [8] wird ein Algorithmus vorgestellt, welcher diese freien Typvariablen als Java Generics generalisiert.

Generics in Java sind wie Typvariablen zu verstehen. Durch sie werden Klassen und Methoden generisch, das heißt, sie können auf verschiedene Typen angewendet werden. Das ist zum Beispiel sinnvoll, um Containerklassen zu implementieren, welche für unterschiedliche Typen verwendet werden können. Ein generischer Parameter kann mit einem Bound versehen werden, was genau der Relation ($T \lessdot S$) entspricht.

Da Generics jeweils auf Methoden und Klassen angewendet werden können,

stellt sich die Frage, wie die Typvariablen generalisiert werden sollen. In [8] wird ein hybrider Ansatz verfolgt, das heißt, Generics werden jeweils auf die Klasse und ihre zugehörigen Methoden verteilt, je nachdem in welchem Kontext die Typvariable verwendet wird. So muss beispielsweise eine Typvariable, die in einem Instanzfeld verwendet wird, automatisch der Klasse zugeordnet werden.

```
1  public class Optional<T> {
2      private final T element;
3
4      public Optional(T element) {
5          this.element = element;
6      }
7
8      public boolean isEmpty() {
9          return element == null;
10     }
11
12     public T get() {
13         if (element == null)
14             throw new NullPointerException();
15         return element;
16     }
17 }
```

Listing 4: Ein Beispiel für einen generischen Container

4.4.1 Familie von generierten Generics

In [8] wird eine Familie von Generics erstellt. Dies geschieht in zwei Schritten. Zunächst werden die Generics für die Klasse erstellt, danach für die einzelnen Methoden. Ausgangspunkt sind hierbei die Constraints, die bei der Typinferenz erstellt werden.

Die Generics der Klasse werden folgendermaßen erstellt:

- Alle Typvariablen der Felder der Klasse mit ihren Bounds
- Die Untergruppe aller Bounds von Typvariablen der Felder mit ihren Bounds
- Alle ungebundenen Typvariablen und ungebundene Bounds mit Object als Bound

Die Implementierung dafür sieht folgendermaßen aus:

```
1  Set<ResultPair<?, ?>> generics(ClassOrInterface
       classOrInterface) {
2      ...
3      Set<ResultPair<?, ?>> result = new HashSet<>();
4      for (var field : classOrInterface.getFieldDecl()) {
5          // Iteriere über sämtliche Felder der Klasse
6          // und finde Bounds dieser Felder
7          findAllBounds(field.getType(), result);
8      }
9      ...
10     return result
11 }
```

```
1  void findAllBounds(RefTypeOrTPHOrWildcardOrGeneric type, Set<
       ResultPair<?, ?>> generics) {
2      if (type instanceof TypePlaceholder tph) {
3          ...
4          var concreteType = concreteTypes.get(tph);
5          if (concreteType != null) {
6              // Falls ein konkreter Typ gefunden wurde (das
                   heißt ein = Constraint)
7              // werden Bounds dieses Typs abgefragt
8              findAllBounds(concreteType, generics);
9              return;
10         }
```

```
11
12          for (var rsp : simplifiedConstraints) {
13              ...
14              if (left.equals(tph)) {
15                  var pair = new PairTPHsmallerTPH(tph, right);
16                  if (!generics.contains(pair)) {
17                      // Bounds des Feldes
18                      generics.add(pair);
19                      // Rekursiver Aufruf zum Finden des
                            Bounds der Bounds
20                      findAllBounds(right, generics);
21                  }
22                  return;
23              }
24          }
25          // Falls keine Bounds gefunden wurden, ist Object der
                Bound
26          generics.add(new PairTPHequalRefTypeOrWildcardType(
                tph, OBJECT));
27      } else if (type instanceof RefType refType) {
28          // Generischer Typ, finde Bounds der Typparameter
29          // Wird nicht verwendet, aber der Vollständigkeit
                halber implementiert
30          refType.getParaList().forEach(t -> findAllBounds(t,
                generics));
31      }
32  }
```

Die Generics der Klasse können noch weiter vereinfacht werden. Laut dem Paper werden innere Typvariablen entfernt, also zum Beispiel Parameter von Lambda-Ausdrücken. Wichtig ist hierbei, dass die Relationen beibehalten werden. Also wenn beispielsweise B in der Relation $(A \lessdot B \lessdot C)$ entfernt wird, muss weiterhin gelten, dass $(A \lessdot C)$. In der Implementierung kam es durch diesen Schritt zu einem Problem, da Lambda-Ausdrücke als Methoden der Klasse implementiert wurden und diese sehr wohl generische Parameter haben können. Gelöst wurde das Problem dadurch, dass die linke Seite der

Relation und die rechte Seite gleichgesetzt wurden. Ob dies allerdings korrekt ist, muss noch überprüft werden.

Eine weitere Vereinfachung ist das Gleichsetzen von Typvariablen in kontravarianter und kovarianter Position. Dies wird bei Funktionstypen interessant. Da der Unify-Algorithmus bereits die Varianz berechnet, ist die Implementierung hier trivial.

Nach der Generierung der Generics für die Klasse müssen in mehreren Schritten die Generics für die einzelnen Methoden generiert werden. Betrachtet wird hierbei die Methode m.

- Typvariablen der Methode m ihren Bounds, wobei diese Bounds auch Typvariablen der Methode sind.
- Die Konstruierten Paare $T_1 \lessdot T_2$ der Typvariablen der Methode, die in dem transitiven Abschluss

  $$T_1 \lessdot R_1 \ll R_2 \lessdot T_2$$

  enthalten sind, wobei $R_1 < R_2$ in den Generics der Methode m' steht und m' in m aufgerufen wird.
- Alle Typvariablen der Methode m mit ihren Bounds, die Typvariablen von Feldern sind.
- Alle ungebundenen Typvariablen der Methode m und alle ungebundenen Bounds mit `Object` als Bound.

4.4.2 Sonderfälle

Wenn nach diesen Regeln die Generics für die Klasse und die Methoden erstellt wurden, müssen noch ein paar Sonderfälle betrachtet werden. Es ist im Allgemeinen nicht möglich, dass zwei Klassen gegenseitig voneinander erben. Die Typinferenz kann allerdings so einen Fall erzeugen. In einem seperaten Schritt müssen diese Zyklen entfernt werden.

```
class Cycle {
    m (x, y) {
        y = x;
        x = y;
    }
}
```

Das Ergebnis bei diesem Beispiel ist $cs'_m = \{(L \lessdot M), (M \lessdot L)\}$.

Für einen Zyklus wird folgender Algorithmus angewendet:

- Alle Typvariablen. die den Zyklus bilden, werden durch eine neue Typvariable X ersetzt.
- Alle Konstraints, die diesen Zyklus gebildet haben, werden entfernt.
- Alle entfernten Typvariablen werden durch X ersetzt.

In der Implementierung des Zyklenfinders wurde der Algorithmus von Tiernan verwendet, der in [9] vorgestellt wird.

Ein zweiter Sonderfall ist das sogenannte Infimum. Dies tritt auf, wenn für einen Typparameter mehrere Bounds definiert sind.

Ein Beispiel für ein solches Infimum ist wie folgt:

```
class Infimum {
    m (x, y, z) {
        y = x;
        z = x;
    }
}
```

Hier sind die Constraints $cs'_m = \{(L \lessdot M), (L \lessdot N), (M \lessdot \texttt{Object}), (N \lessdot \texttt{Object})\}$

Diese Infima werden mit folgendem Algorithmus entfernt, T ist dabei die linke Seite der Regeln, die zum Infimum gehört:

- Erstelle eine neue Typvariable X und ein Constraint $(T \lessdot X)$.

- Füge dieses Constraint in die Liste der Constraints ein.
- Entferne alle anderen Constraints von X.
- Alle rechten Seiten dieser Constraints werden auf X abgebildet.

4.4.3 Beispiel

Hier eingefügt ist ein Beispiel, welches in [8] aufgeführt wird und sehr vollständig ist.

```
1  class TPHsAndGenerics {
2      id = x -> x;
3
4      id2 (x) {
5          return id.apply(x);
6      }
7
8      m (a, b) {
9          var c = m2(a, b);
10         return a;
11     }
12
13     m2 (a, b) {
14         return b;
15     }
16 }
```

```
1  class TPHsAndGenerics {
2      Fun1$$<UD, ETX> id = DZP x
           -> x;
3
4      ETX id2 (V x) {
5          return id.apply(x);
6      }
7
8      AB m (AB a, AD b) {
9          AE c = m2(a, b);
10         return a;
11     }
12
13     AI m2 (AM a, AI b) {
14         return b;
15     }
16 }
```

Links steht der JavaTX Code und rechts der generierte AST mit Typinformationen. Die vom Unify generierten Constraints sehen folgendermaßen aus:

$$cs = \{\texttt{AD} \lessdot \texttt{AI}, \texttt{V} \lessdot \texttt{UD}, \texttt{AI} \lessdot \texttt{AE}, \texttt{AB} \lessdot \texttt{AM}, \texttt{DZP} \lessdot \texttt{ETX}, \texttt{UD} \lessdot \texttt{DZP}\}$$

Das Endergebnis der generierten Generics sieht wie folgt aus:

```
1  class TPHsAndGenerics<UD> {
2      Fun1$$<UD, UD> id = x -> x;
3
4      <V extends UD> UD id2 (V x) {
5          return id.apply(x);
6      }
7
8      <AD extends AE, AB, AE> AB m (AB a, AD b) {
9          AE c = m2(a, b);
10         return a;
11     }
12
13     <AI, AM> AI m2 (AM a, AI b) {
14         return b;
15     }
16 }
```

4.5 Implementierung des Bytecode-Generierers

Die Bytecode-Generierung wurde mit Hilfe des Tools ASM geschrieben, das schon bei der letzten Iteration verwendet worden ist. Die Vorteile hierbei sind:

- Die verschiedenen Codes werden mit Hilfe von Funktionen erzeugt, die jeweils die Argumente aus den Parametern berechnen.
- Der Konstantenpool wird aus den im Bytecode verwendeten Konstanten generiert.
- Sprünge werden mit Hilfe von Labels erzeugt. Es muss nicht die Adresse des Sprungs angegeben werden.
- Metadata, wie die maximale Stack-Größe, wird automatisch berechnet.
- Die Umwandlung in binäre Class-Dateien wird von ASM durchgeführt.

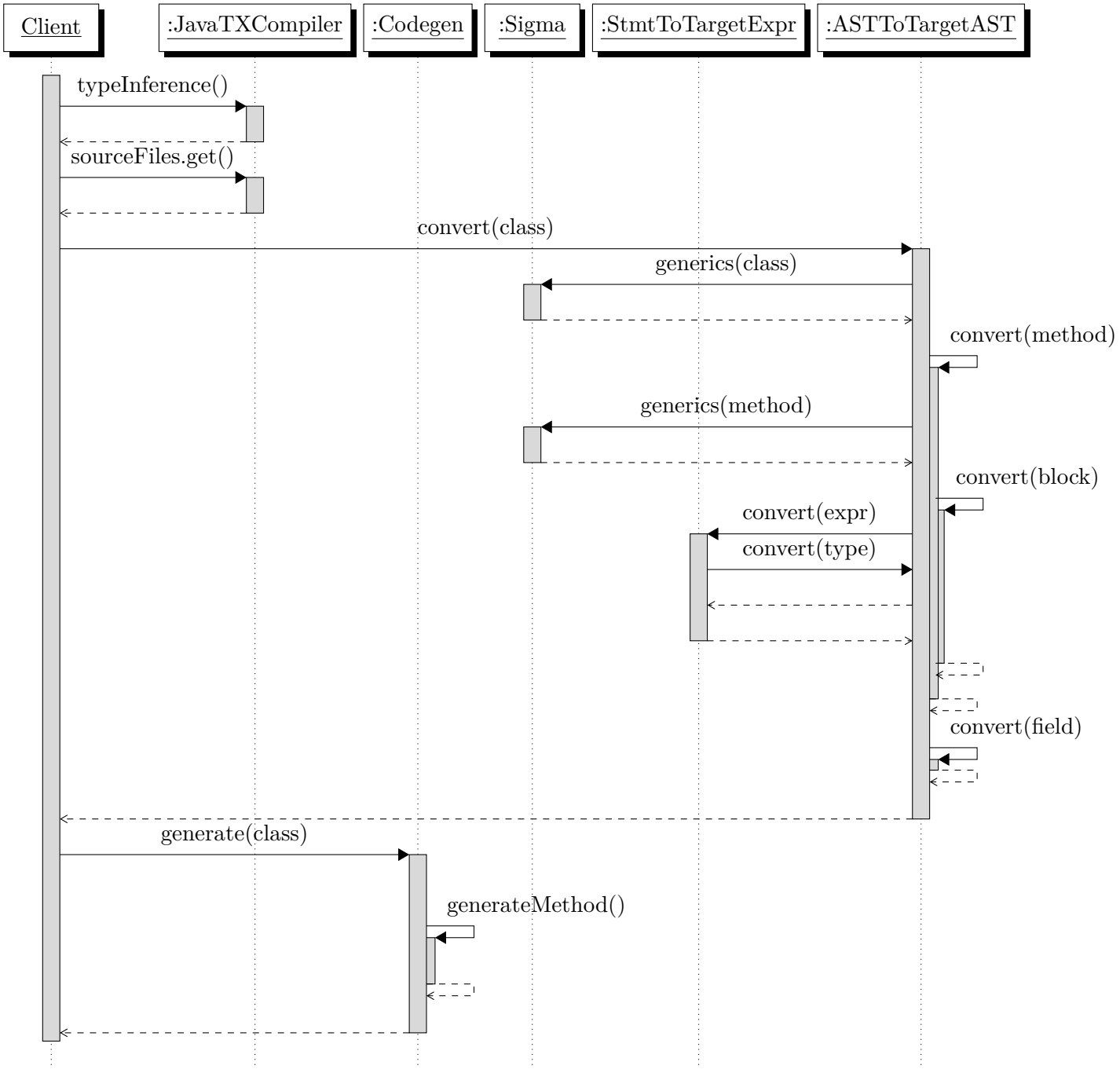

Abbildung 2: Sequenzdiagramm Codegenerierer

4.5.1 Variablen

In Java gibt es mehrere Arten von Variablen, die jeweils im Bytecode anders repräsentiert sind. Da wären zum einen die Instanzvariablen. Diese werden jeweils der Objektinstanz zugeordnet und mit dem Opcode `PUTFIELD/GETFIELD` adressiert. Beim Aufruf dieses Opcodes muss auf dem Stack eine Objektinstanz liegen, zu welcher die Variable gehört. Als nächstes gibt es statische Variablen, die jeweils der Klasse zugeordnet werden. Diese werden mit dem Opcode `PUTSTATIC/GETSTATIC` adressiert. Da diese zwei Typen ähnlich aufgebaut sind, werden sie im Target-AST durch dasselbe Konstrukt repräsentiert: `TargetFieldVar`.

Die nächsten zwei Typen sind Parameter und lokale Variablen, die allerdings im Bytecode genau gleich repräsentiert werden. Hierzu gibt es verschiedene Befehle, je nach Typ der Variable. In JavaTX werden nur Referenztypen verwendet, das heißt, im Falle von primitiven Datentypen die dazugehörigen Wrapper. Daswegen wird nur jeweils `ASTORE/ALOAD` verwendet. Wichtig ist hierbei, dass lokale Variablen, anders als Felder, durch einen Index angesprochen werden. Um das zu realisieren, wurde eine neue Klasse `Scope` erstellt. Ein Scope enthält eine `HashMap`, welche den Variablennamen auf einen Index abbildet. Dieser Index startet bei 1 (Index 0 entspricht der `this`-Instanz) und wird beim Verarbeiten einer `TargetVarDecl` inkrementiert.

```
1  // Zuweisungen werden durch TargetAssign repräsentiert
2  case TargetAssign assign: {
3      switch (assign.left()) {
4          case TargetLocalVar localVar: {
5              generate(state, assign.right());
6              boxPrimitive(state, assign.right().type());
7              var local = state.scope.get(localVar.name());
8              mv.visitInsn(DUP);
9              mv.visitVarInsn(ASTORE, local.index());
10             break;
11         }
```

```
12          case TargetFieldVar dot: {
13              generate(state, dot.left());
14              generate(state, assign.right());
15              boxPrimitive(state, assign.right().type());
16              if (dot.isStatic())
17                  mv.visitInsn(DUP);
18              else mv.visitInsn(DUP_X1);
19              mv.visitFieldInsn(dot.isStatic() ? PUTSTATIC :
                    PUTFIELD, dot.owner().getInternalName(), dot.
                    right(), dot.type().toSignature());
20              break;
21          }
22          default:
23              throw new CodeGenException("Invalid assignment");
24      }
25      break;
26  }
27  case TargetLocalVar localVar: {
28      LocalVar local = state.scope.get(localVar.name());
29      mv.visitVarInsn(ALOAD, local.index());
30      unboxPrimitive(state, local.type());
31      break;
32  }
33  case TargetFieldVar dot: {
34      if (!dot.isStatic())
35          generate(state, dot.left());
36      mv.visitFieldInsn(dot.isStatic() ? GETSTATIC : GETFIELD,
            dot.left().type().getInternalName(), dot.right(), dot.
            type().toSignature());
37      unboxPrimitive(state, dot.type());
38      break;
39  }
```

Listing 5: Implementierung der Variablen

4.5.2 Methoden und Konstruktoren

Methoden werden in ASM durch Aufruf eines `MethodVisitor`s generiert. Ein Konstruktor ist lediglich eine spezielle Methode mit dem Namen `<init>`. Deshalb werden beide ähnlich behandelt. In Java können Methoden nach ihren Parametern überladen werden. In JavaTX werden die Überladungen automatisch generiert, und zwar je nachdem welche Typen für die Parameter ausgewählt werden. Dies wird dadurch realisiert, dass im `ResultSet` mehrere Lösungen für die Typvariablen hinterlegt sind. Damit es nicht zu einer Explosion von Möglichkeiten kommt, und dadurch die Laufzeit exponentiell steigt, müssen die Klassen, welche für eine Überladung ausgewählt werden, vorher importiert werden.

Um mehrere Methoden zu generieren, werden bei der Konvertierung jeweils alle Lösungen aus dem `ResultSet` betrachtet und eine neue `TargetMethod` generiert, falls die Parameter in dieser Form noch nicht vorgekommen sind.

```
1 // JavaTX
2 import java.lang.Integer;
3 import java.lang.String;
4
5 public class Plus {
6     m (a,b) {
7         return a + b;
8     }
9 }
```

```
1  // Dekompilierte Klasse
2  public class Plus {
3      public String m(String var1, String var2) {
4          return var1 + var2;
5      }
6
7      public Integer m(Integer var1, Integer var2) {
8          return var1 + var2;
9      }
10 }
```

Listing 6: Überladung der Methode m

4.5.3 Operatoren

In Java gibt es eine Vielzahl von arithmetischen, logischen und relationalen Operatoren. Da es in Java keine Operatorüberladung gibt, sind Operatoren relativ einfach zu implementieren. Die Implementierung ist jeweils ähnlich, da es Befehle auf der Bytecodeebene gibt, welche diesen Operatoren entsprechen. Bei den binären Operatoren werden zunächst beide Operanten auf den Stack gelegt. Beide werden in ihre primitiven Formen umgewandelt, siehe 4.5.5. Dann wird der Befehl je nach Ausgangstyp der beiden Parameter ausgewählt.

```
1  case Sub sub: {
2      generate(state, sub.left());
3      convertTo(state, sub.left().type(), op.type());
4      generate(state, sub.right());
5      convertTo(state, sub.right().type(), op.type());
6      var type = sub.type();
7      if (type.equals(TargetType.Byte)
8              || type.equals(TargetType.Char)
9              || type.equals(TargetType.Integer)
10             || type.equals(TargetType.Short)) {
```

```
11          mv.visitInsn(ISUB);
12      } else if (type.equals(TargetType.Long)) {
13          mv.visitInsn(LSUB);
14      } else if (type.equals(TargetType.Float)) {
15          mv.visitInsn(FSUB);
16      } else if (type.equals(TargetType.Double)) {
17          mv.visitInsn(DSUB);
18      } else {
19          throw new CodeGenException("Invalid argument to Sub
                expression");
20      }
21      break;
22  }
```

Listing 7: Implementierung der Subtraktion

4.5.4 Kontrollstrukturen

Kontrollstrukturen wie das If-Statement und die For-Schleife sind auf Bytecodeebene mit Sprungbefehlen umgesetzt. Das Tool ASM besitzt für diese eine Label-Implementierung die es ermöglicht, ohne die explizite Verwendung von numerischen Adressen, Sprünge zu generieren. Der Oder und der Und-Operator benötigen auch Sprünge, da beide kurzschließend sind. Das heißt, die linke Seite des Operators wird zuerst betrachtet und wenn dieser Ausdruck zu Falsch (Beim Und-Operator) beziehungsweise Wahr (beim Oder-Operator) ausgewertet wird, wird die rechte Seite des Operators nicht mehr betrachtet.

Nun soll zunächst die klassische For-Schleife betrachtet werden. In Java ist diese folgendermaßen aufgebaut:

```
1  // Initialisierung; Abbruchbedingung; Iteration
2  for (var i = 0; i < 10; i++) {
3      // Rumpf
4  }
```

Man kann diese For-Schleife auch durch eine einfacherere While-Schleife ersetzen:

```
1 // Initialisierung
2 var i = 0;
3 // Abbruchbedingung
4 while (i < 10) {
5     // Rumpf
6     // Iteration
7     i++;
8 }
```

Hier wird recht gut deutlich, dass insgesamt zwei Labels benötigt werden um diese Schleife umzusetzen. Ein Label markiert den Beginn der Schleife und ein weiteres wird an das Ende der Schleife eingefügt.

Die Initialisierung wird zuerst ausgeführt. Wenn die Abbruchbedingung Wahr ist, wird zunächst der Rumpf der Schleife ausgeführt. Danach wird die Iteration ausgeführt. Nun wird zurück zum Anfang der Schleife gesprungen. Falls die Abbruchbedingung Falsch ist, wird die Schleife terminiert.

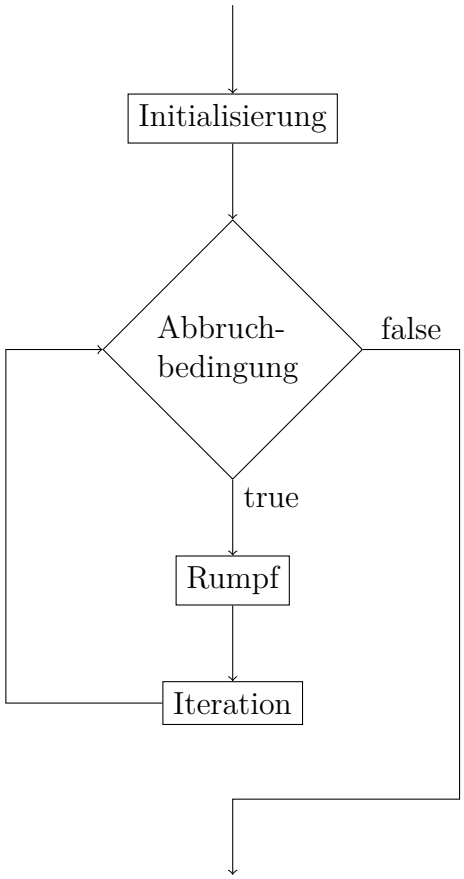

Abbildung 3: For-Schleife Flussdiagram

Die Umsetzung im Bytecode-Generierer sieht folgendermaßen aus:

```
1 case TargetFor _for: {
2     state.enterScope();
3     var localCounter = state.localCounter;
4     // Der Initialsierer, die Abbruchbedingung und auch der
        Iterationsschritt können alle leer sein, deshalb wird
        nach null überprüft.
5     if (_for.init() != null)
6         generate(state, _for.init());
7
8     Label start = new Label();
```

```
9       Label end = new Label();
10
11      mv.visitLabel(start);
12      if (_for.termination() != null)
13          generate(state, _for.termination());
14      else mv.visitInsn(ICONST_1);
15      // Falls keine Abbruchbedingung vorhanden ist, wird 1 (
            Wahr) auf den Stack gelegt.
16
17      // IFEQ springt an das Ende der Schleife falls die
            Bedingung Falsch ergibt.
18      mv.visitJumpInsn(IFEQ, end);
19
20      // Rumpf der For-Schleife
21      generate(state, _for.body());
22      if (_for.increment() != null) {
23          generate(state, _for.increment());
24          if (_for.increment().type() != null) {
25              mv.visitInsn(POP);
26          }
27      }
28      mv.visitJumpInsn(GOTO, start);
29      mv.visitLabel(end);
30      state.exitScope();
31      state.localCounter = localCounter;
32      break;
33  }
```

4.5.5 Autoboxing

In Java gibt es eine Trennung von primitiven Datentypen und Referenztypen. Primitve Datentypen sind beispielsweise `int`, `float` oder `double`. Das Problem mit primitiven Datentypen ist, dass sie nicht als generischer Parameter verwendet werden können. Das liegt daran, dass alle generischen Typen zur Laufzeit vom Typ `Object` sind und primitive Datentypen aus Performancegründen nicht Teil der Objekthierarchie sind. Als Ersatz für diese primitiven

Typen gibt es sogenannte Wrapper-Typen wie `Integer`. Diese erben von `Object` und können deshalb als generischer Typ verwendet werden. Damit diese Typen einfach verwendet werden können, gibt es in Java Autoboxing. Das heißt, primitive Datentypen werden automatisch in ihre Wrapper umgewandelt und umgekehrt.

```
1     // JavaTX
2     // Primitiver Datentyp int
3     int a = 20;
4     // Wrapper-Typ Integer, int konvertiert automatisch zu
          Integer
5     Integer b = a;
6     ArrayList<Integer> list1;
7     // Wird nicht kompiliert da int ein primitiver Datentyp
          ist
8     ArrayList<int> list2;
```

In JavaTX werden nur die Wrapper-Typen verwendet um die Typinferenz zu vereinfachen. Sonst müssten immer Überladungen für mehrere Typen generiert werden. Allerdings müssen die primitiven Typen im generierten Bytecode verwendet werden, da Operatoren wie Addition oder Subtraktion nur auf primitiven Datentypen funktionieren. Ebenfalls benötigt werden diese primitiven Typen um Funktionen aufzurufen, die außerhalb von JavaTX erstellt wurden. Deshalb besitzt der Target-AST jeweils für beide Typen eine Repräsentation.

Die Implementierung besitzt demnach zwei Funktionen um einen Typ zu boxen beziehungsweise um ihn zu unboxen.

```
1 private void boxPrimitive(State state, TargetType type) {
2     var mv = state.mv;
3     if (type.equals(TargetType.Boolean) || type.equals(
          TargetType.boolean_)) {
```

```
4           // Es wir jeweils die Methode valueOf aufgerufen, je
                nach Typ der übergeben wird.
5           // Dieser Branch wird sowohl beim primitiven Typ als
                auch beim Wrapper-Typ ausgeführt.
6           mv.visitMethodInsn(INVOKESTATIC, "java/lang/Boolean",
                 "valueOf", "(Z)Ljava/lang/Boolean;", false);
7       } else if (type.equals(TargetType.Byte) || type.equals(
            TargetType.byte_)) {
8           mv.visitMethodInsn(INVOKESTATIC, "java/lang/Byte", "
                valueOf", "(B)Ljava/lang/Byte;", false);
9       }
10      ...
11  }
12
13  private void unboxPrimitive(State state, TargetType type) {
14      var mv = state.mv;
15      if (type.equals(TargetType.Boolean)) {
16          // Beim unboxen wird jeweils die Instanzmethode
                aufgerufen, die
17          // den Wrappertyp in den equivalenten primitiven Typ
                umwandelt.
18          mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean"
                , "booleanValue", "()Z", false);
19      } else if (type.equals(TargetType.Byte)) {
20          mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Byte", "
                byteValue", "()B", false);
21      }
22      ...
23  }
```

Die Funktion `unboxPrimitive` wird immer dann aufgerufen, wenn ein primitiver Datentyp für eine Kalkulation gebraucht wird. Das heißt, zum Beispiel beim Aufruf eines Operators oder beim Aufrufen einer Funktion, die einen primitiven Datentyp erwartet. Wenn diese Operation ausgeführt wurde, wird also `boxPrimitive` aufgerufen, damit der Typ auf dem Stack immer der geboxte Typ ist. Das ist wichtig, da die Variablen in JavaTX, wie eingangs

erwähnt, nur die Wrapper als Typ haben, also allesamt `ALOAD/ASTORE` benutzen. Die Implementierung hierfür ist sehr konservativ. Das heißt, es wird immer eine Konvertierung benutzt, auch wenn diese nicht unbedingt nötig ist. Wenn also zum Beispiel mehrere Werte addiert werden, wird zwischendurch immer wieder in den Wrapper-Typ umgewandelt. Hier wäre es also noch möglich den Codegenerierer zu optimieren.

4.5.6 Lambda-Ausdrücke

Lambda-Ausdrücke werden in funktionellen Programmiersprachen Closures genannt. Sie sind eine realtiv neue Addition und wurden erst in Java-8 eingeführt. Davor war es nötig, innere Klassen zu verwenden, um dieselbe Funktionalität zu erreichen. Die Implementierung von Lambda-Ausdrücken in Java ist recht speziell. Um einen Lambda-Ausdruck zu verwenden, muss als Typ ein sogenanntes funktionales Interface benutzt werden. Dies sind Interfaces mit genau einer Methode, die dann von dem Lambda-Ausdruck implementiert wird. Das ist deshalb sinnvoll, da bereits viele in Java vorkommende Interfaces bereits funktionale Interfaces sind und deshalb ohne Anpassungen von Lambda-Ausdrücken implementiert werden können. Ein Beispiel dafür ist das `Runnable` Interface mit dem neue Threads gestartet werden:

```
1  public static void main(String[] args) {
2      // Java 7
3      Thread thread = new Thread(new MyRunnable());
4      thread.start();
5
6      // Java 8
7      Thread thread2 = new Thread(() -> {
8          System.out.println("Hello from another Thread");
9      })
10     thread2.start();
11 }
12
13 public static class MyRunnable implements Runnable {
```

```
14      @Override
15      public void run() {
16          System.out.println("Hello from another Thread");
17      }
18  }
```

Allerdings gibt es hier mehrere Einschränkungen. So hat der Lambda-Ausdruck selbst keinen Typ und kann deshalb nicht mit der Typinferenz aus Java 9 verwendet werden. Das zweite Problem hat mit der Typeerasure zu tun. Es wäre sinnvoll, wenn in der Java Standardbibliothek die Funktionstypen bereits voll implementiert werden. Es ist aber nicht möglich, dieselbe Klasse mit unterschiedlich vielen Typparametern zu definieren. Deshalb kann man nicht die naheliegende Implementierung mit einem Typ `Function` verwenden. `Function<Return, Param1>` und `Function<Return, Param1, Param2>` sind also nicht möglich. Statt dessen werden in der Java Standardbibliothek mehrere häufig genutzten Funktionstypen mit unterschiedlichen Namen definiert, wie beispielsweise `Consumer`, `Function` und `Predicate`. Das kann sehr verwirrend sein und ist deshalb von anderen Sprachen, die auf der JVM basieren, anders gelöst worden. Scala beispielsweise besitzt eine eigene Syntax für Funktionstypen.

```
1  // Funktioniert nicht, Typ von add2 kann nicht inferiert
       werden
2  var add2 = (int a) -> a + 2;
3  // Funktioniert, Lambda bekommt den Typ IntFunction
4  IntFunction<Integer> add2 = (a) -> a + 2;
```

In JavaTX wurden ebenfalls eigene Typen für Funktionen definiert. Dadurch ist es möglich, diese inferieren zu lassen. Diese Funktionen haben die Form `FunN$$`. Diese müssen allerdings nicht direkt benutzt werden, sondern werden automatisch inferiert.

Obwohl eine gewisse Ähnlichkeit zu inneren Klassen besteht, werden Lambda-Ausdrücke anders implementiert. Zunächst muss für den Rumpf des Lambda-

Ausdrucks eine Methode generiert werden. Dies kann je nach Kontext entweder eine statische oder eine Instanzmethode sein. Da JavaTX nach jetzigem Stand noch keine statischen Methoden kennt, wird immer eine Instanzmethode generiert. Das ist nützlich, da so über den `this`-Pointer auf die umschließende Klasse zugegriffen werden kann. Das heißt, der Code des Lambda-Ausdrucks kann genau so verwendet werden, wie er bereits definiert worden ist.

Wenn im Lambda-Ausdruck Variablen des umschließenden Scopes verwendet werden, müssen diese als Parameter übergeben werden. Wichtig hierbei ist, dass diese Variablen nicht zugewiesen werden können, was die Implementierung stark vereinfacht. Um diese eingefangenen Variablen zu finden, wird über den gesamten Ausdruck iteriert und ein `Stack` für die Variablen, welche im Lambda-Ausdruck definiert werden, erstellt. Alle Variablen. die hierbei übrig bleiben, kommen demnach aus dem umschließenden Scope und müssen als Parameter übergeben werden.

Im Bytecode werden Lambda-Ausdrücke mit dem Befehl `INVOKEDYNMAIC` erstellt. Die Parameter hierbei sind ähnlich wie bei dem Aufrufen einer Methode. Zusätzlich wird noch ein Handle für eine `LambdaMetafactory`, welche die Instanz des funktionalen Interfaces aus der übergebenen Methode erstellt. Dafür wird zur Laufzeit eine Klasse erstellt, welche die Methode des funktionalen Interfaces implementiert.

```
1 ...
2 TargetMethod impl = ...;
3 // Erstellen der Metadata für den INVOKEDYNMAIC Aufruf
4 var mt = MethodType.methodType(CallSite.class, MethodHandles.
      Lookup.class, String.class,
5      MethodType.class, MethodType.class, MethodHandle.class,
          MethodType.class);
6
7 var bootstrap = new Handle(H_INVOKESTATIC, "java/lang/invoke/
      LambdaMetafactory", "metafactory",
```

```
8           mt.toMethodDescriptorString(), false);
9   var handle = new Handle(
10          H_INVOKEVIRTUAL, clazz.getName(), impl.name(),
11          impl.getDescriptor(), false
12  );
13  ...
14  // INVOKEDYNMAIC
15  mv.visitInvokeDynamicInsn("apply", descriptor,
16      bootstrap, Type.getType(desugared), handle,
17      Type.getType(TargetMethod.getDescriptor(impl.returnType()
            , lambda.params().stream().map(MethodParameter::type).
            toArray(TargetType[]::new)))
18  );
```

4.5.7 Signaturen

Eine Signatur enthält die Typinformationen für eine Methode, Klasse oder ein Feld. In Java gibt es neben der eigentlichen Signatur auch noch einen Deskriptor. Während die Signatur auch generische Typen enthält, besteht der Deskriptor nur aus zur Laufzeit tatsächlich verwendeten Typen. Ein generischer Typ wird demnach durch seinen Bound oder `Object` ersetzt. Der Deskriptor ist bereits älter und seit den ersten Versionen von Java vertreten. Die Signatur gibt es seit Java 5, der Version mit der generische Parameter eingeführt wurden.

```
1   public List<Integer> rangeOfNumbers(int start, int end);
2   // Deskriptor:
3   // (II)Ljava/util/List;
4   // Signatur:
5   // (II)Ljava/util/List<Ljava/lang/Integer;>;
6
7   public <T extends Number> List<T> rangeOf(T start, T end);
8   // Deskriptor:
9   // (Ljava/lang/Number;Ljava/lang/Number;)Ljava/util/List;
10  // Signatur:
```

`boolean`	`Z`
`byte`	`B`
`int`	`I`
`long`	`J`
`float`	`F`
`double`	`D`
`char`	`C`
`short`	`S`
`int[]`	`[I`
`java.util.List`	`Ljava/util/List;`

Tabelle 1: Deskriptor von Primitiv- und Referenztypen

```
11 // <T:Ljava/lang/Number;>(TT;TT;)Ljava/util/List<TT;>;
```

Listing 8: Beispiel einer Signatur und Deskriptor

Primitive Typen werden durch einzelne Zeichen ersetzt, während Referenztypen mit ihrem Package und dem Klassennamen angesprochen werden.

Typparameter einer Methode werden in der Signatur vor den Parametern in spitzen Klammern angegeben. Typparameter eines Referenztyps werden nach dem Deskriptor ebenfalls in spitzen Klammern angegeben.

4.5.8 Unittests

Die Tests für den Codegenerierer wurden mit der Bibliothek JUnit erstellt. JUnit stellt den Testrunner und verschiedene Assertions zu Verfügung. Test-Methoden werden dabei mit der Annotation `org.junit.Test` versehen. Ein großer Vorteil von JUnit ist, dass die Integration mit der Entwicklungsumgebung sehr gut ist. Tests können in IDEA über ein seperates Panel ausgeführt werden und Informationen zu den Tests wie die Laufzeit oder den Status des Tests (Erfolgreich/Fehler) werden angezeigt.

Bei den Tests handelt es sich vor allem um Integrationstests, das heißt es werden sämtliche Komponenten des Compilers auf Fehler überprüft. Ebenfalls gibt es unter der Klasse `TestCodegen` mehrere Tests. die manuell einen Target-AST erstellen und diesen an der Codegenerierer übergeben. Hier wird

also der Codegenerierer seperat getestet. Der Aufbau vom Target-AST wird über die Konstruktoren, welche für die Rekordtypen generiert wurden, bewerkstelligt.

```
1  @Test
2  public void testIfStatement() throws Exception {
3      // Zunächst wird eine neue Klasse erstellt
4      var targetClass = new TargetClass(Opcodes.ACC_PUBLIC, "
           IfStmt");
5      // Eine Methode "ifStmt" wird hinzugefügt
6      // Der Methodenrumpf wird als TargetBlock übergeben
7      // (Hier verkürzt dargestellt)
8      targetClass.addMethod(Opcodes.ACC_PUBLIC | Opcodes.
           ACC_STATIC, "ifStmt",
9          List.of(new MethodParameter(TargetType.Integer, "val"
               )),
10         TargetType.Integer,
11         new TargetBlock(List.of(new TargetIf(...)))
12     );
13
14     // Hier wird die Klasse durch einen ClassLoader geladen
15     // und über die Reflection-API angesprochen
16     var clazz = generateClass(targetClass, new
           ByteArrayClassLoader());
17     var ifStmt = clazz.getDeclaredMethod("ifStmt", Integer.
           class);
18
19     // Test Assertions
20     assertEquals(ifStmt.invoke(null, 10), 1);
21     assertEquals(ifStmt.invoke(null, 3), 2);
22     assertEquals(ifStmt.invoke(null, 20), 3);
23 }
```

Listing 9: Test des Codegenerieres für If-Statements

Das Testen der generierten Generics erweist sich als etwas komplizierter. Wie bei den anderen Tests auch, wird die generierte Klasse durch einen `ClassLoader` geladen. Das Problem ist nun, dass es einen gewissen Nichtdeterminismus gibt. Die Namen der Typparameter sind nicht fest und auch ihre Reihenfolge kann sich von Aufruf zu Aufruf ändern. Es gibt allerdings noch eine weitere Möglichkeit die Typparameter zu testen. Dafür wird, von den Typen der übergebenen Parameter der Methode ausgehend, auf die Existenz und Gleichheit verschiedener Typparameter getestet.

```
1  @Test
2  public void tph2Test() throws Exception {
3      var classFiles = generateClassFiles("Tph2.jav", new
           ByteArrayClassLoader());
4      var tph2 = classFiles.get("Tph2");
5      var instance = tph2.getDeclaredConstructor().newInstance
           ();
6
7      assertEquals(1, tph2.getTypeParameters().length);
8      // public class Tph2<DZG>
9      var DZG = tph2.getTypeParameters()[0];
10
11     var id = tph2.getDeclaredField("id");
12     // public Fun1$$<DZG, DZG> id
13     var idParams = ((ParameterizedType) id.getGenericType()).
           getActualTypeArguments();
14     assertEquals(2, idParams.length);
15     assertEquals(DZG, idParams[0]);
16     assertEquals(DZG, idParams[1]);
17
18     var id3 = tph2.getDeclaredMethod("id3", Object.class);
19     // public <U extends DZG> DZG id3(U var1)
20     var paraTypes = id3.getGenericParameterTypes();
21     var typeParaTypes = id3.getTypeParameters();
22
23     var U = Arrays.stream(typeParaTypes).filter(t -> t.equals
```

```
            (paraTypes[0])).findFirst().get();
24      assertEquals(DZG, U.getBounds()[0]);
25      assertEquals(DZG, id3.getGenericReturnType());
26  }
```

Listing 10: Test der generischen Parameter von Tph2.jav

5 Ergebnis

5.1 Entwurfsmuster

In der Neuimplementierung des Codegenerieres wurden vor allem zwei Entwurfsmuster benutzt, das Visitor-Pattern und der Typswitch aus Java 18. Im Folgenden sollen nun beide Konzepte miteinander verglichen werden.

5.1.1 Visitor

Das Visitor-Pattern wird im Detail in [4] vorgestellt. Es kann immer dann eingesetzt werden, wenn eine Funktionalität für mehrere Klassen implementiert werden soll, ohne die Klassen selbst zu verändern. Das kann sinnvoll sein, wenn man nicht von vorhinein weiß, was für Operationen gebraucht werden. Außerdem kann so Code, der konzeptionell zusammengehört, in einer Klasse zusammengefasst werden, anstatt auf mehrere Klassen verteilt zu sein.

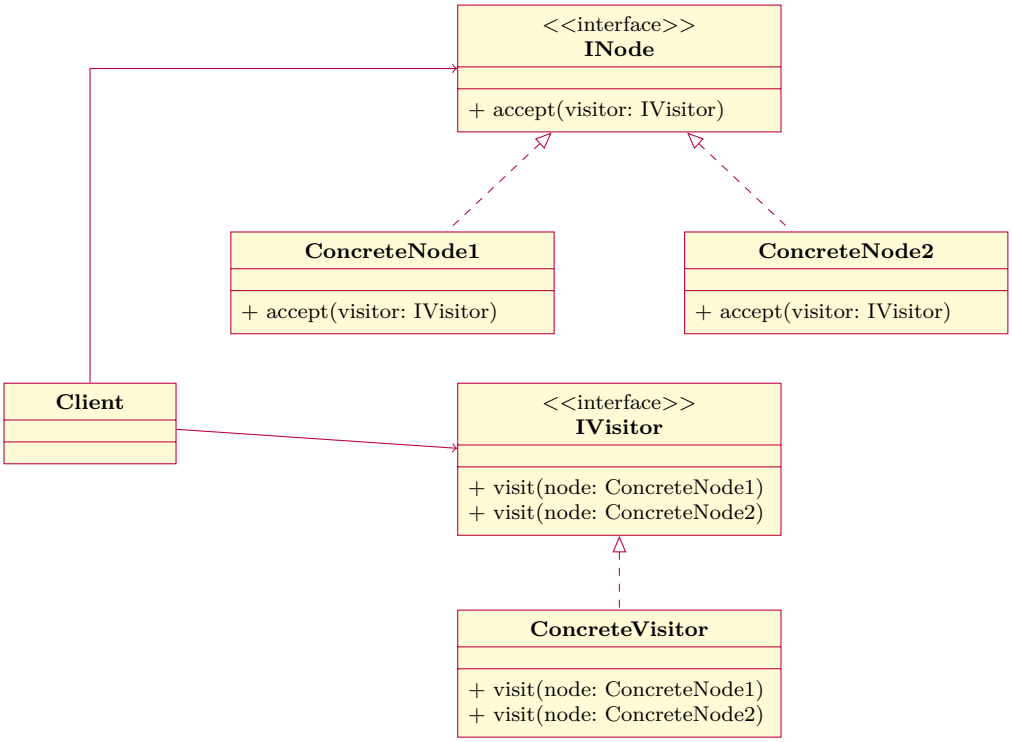

Abbildung 4: UML Diagramm für das Visitor-Pattern

In der Praxis wird der Visitor jeweils von der `accept` Methode der konkreten Node-Klasse aufgerufen. Dieser Vorgang wird auch als Double-Dispatch bezeichnet, da sowohl die Node selber als auch der Visitor für den Aufruf zuständig sind.

Ein Visistor wird meistens auf eine größere Datenstruktur angewendet. Im Falle des Compilers ist das der AST. Um alle Nodes aufzurufen, gibt es hierbei zwei verschiedene Möglichkeiten. Zum einen kann von außerhalb des Visitors für alle Nodes die `accept` Methode aufgerufen werden, also beispielsweise durch eine for-Schleife. Das ist besonders dann sinnvoll, wenn es sich um eine flache Datenstruktur handelt, wie einen Array oder eine Liste. Da der AST aber ein Baum ist, bei dem eine Node unterschiedlich viele Kinder hat, macht es in diesem Fall Sinn, die Iteration des AST über den Visitor zu gestalten. Das bedeutet, der Visitor ruft auf alle Kinder der Node die `accept` Methode auf. Ein Vorteil dieser Vorgehensweise ist, dass es möglich ist, nur jeweils

bestimmte Zweige des Baums aufzurufen, je nach Problemstellung. Auch kann die Art der Traversierung geändert werden, also Preorder, Postorder oder Inorder.

Ein weiterer Vorteil ist, dass die Iteration des Baums getrennt werden kann von der eigentlichen Aufgabe. Das ist im Codegenerierer von der Klasse `TracingStatementVisitor` so gemacht worden. Diese Klasse kann dann von einem konkreten Visitor beerbt werden und es muss nur noch die passende `super` Methode aufgerufen werden, um den Baum zu traversieren. Das ist beim Typswitch nicht möglich. Die Vererbung vereinfacht die Implementierung, auch wenn nur wenige der Node-Klassen tasächlich betrachtet werden sollen. Hier kann entweder eine abstrakte Klasse helfen, oder `default`-Methoden im Visitor-Interface.

Der Visitor und die `accept` Methoden können auch einen Rückgabetyp haben. Das ist dann hilfreich, wenn für eine Operation ein Wert für den gesamten Baum aggregiert werden soll. Der Rückgabetyp kann auch generisch sein, um denselben Visitor für mehrere Operationen zu verwenden.

Mit Hilfe des Visitor-Patterns kann ein einfacher mathematischer Ausdruck ausgewertet werden. Das selbe Beispiel wird später für den Typswitch angepasst.

```
1  interface Node {
2      int accept(Visitor v);
3  }
4  interface Visitor {
5      int visit(Value v);
6      int visit(Add add);
7  }
8
9  class Value implements Node {
10     final int value;
11     Value(int value) {
12         this.value = value;
```

```
13      }
14
15      public int accept(Visitor v) {
16          return v.visit(this);
17      }
18  }
19
20  class Add implements Node {
21      final Node left, right;
22      Add(Node left, Node right) {
23          this.left = left;
24          this.right = right;
25      }
26
27      public int accept(Visitor v) {
28          return v.visit(this);
29      }
30  }
31
32  class Eval implements Visitor {
33      public int visit(Value v) { return v.value; }
34      public int visit(Add a) {
35          return a.left.accept(this) + a.right.accept(this);
36      }
37  }
```

Listing 11: AST-Evaluierer umgesetzt in Java 8

5.1.2 Typswitch

Der Typswitch ist ein neues Feature und wurde in Java 18 eingeführt [2]. Er ist Teil eines größeren Projektes, an dessen Ende Dekonstruktion von beliebigen Klassen über das Switch-Statement beziehungsweise der Switch-Expression stehen. In früheren Versionen von Java war es nur auf umständliche Weise möglich, einen Ausdruck nach mehreren Typen zu überprüfen. Hierzu war es nötig, ein größeres If-Statement mit mehreren Branches zu

schreiben, zusammen mit Typcasts, um Zugriff auf den benötigten Typ zu bekommen. Das ist generell umständlich und nicht zu empfehlen, wenn man die Typhierarchie beeinflussen kann. Das Problem lässt sich auch über Vererbung und dynamische Bindung in einem objektorientierten Ansatz lösen. Eine Alternative dazu ist das eingangs vorgestellte Visitor-Pattern.

Wenn man allerdings keinen Zugriff auf die Typhierarchie hat und zum Beispiel Verbundtypen mit mehreren nicht miteinander verwandten Typen erstellen will, gab es bislang keine andere Möglichkeit.

```
1  Object obj = ...; // Object als übergeordneter Typ
2  double result;
3  if (obj instanceof Integer) {
4      Integer i = (Integer) obj;
5      result = i * 2;
6  } else if (obj instanceof Double) {
7      Float d = (Double) obj;
8      result = d + 2.5d;
9  } else {
10     result = 10d;
11 }
```

Java 16 hat die Situation mit Typmustern in instanceof-Ausdrücken etwas verbessert. Damit ist folgender Code möglich:

```
1  Object obj = ...;
2  double result;
3  if (obj instanceof Integer i) {
4      result = i * 2;
5  } else if (obj instanceof Double d) {
6      result = d + 2.5d;
7  } else {
8      result = 10d;
9  }
```

Der Typcast entfällt demnach. Durch Switch-Expressions (Java 12) und den neu eingeführten Typmustern lässt sich der Code nun nochmal vereinfachen.

```
1 Object obj = ...;
2 double result = switch(obj) {
3     case Integer i -> i * 2;
4     case Double d -> d + 2.5d;
5     default -> 10d;
6 }
```

Interessant ist hierbei, dass nun ein ähnliches Konzept umgesetzt werden kann wie durch das Visitor-Pattern, nämlich eine Fallunterscheidung, bei der über eine bestimmte Menge von Typen abstrahiert wird. Eine weitere Eigenschaft des Visitor-Patterns kann mit Hilfe von Sealed Interfaces (3.2) erreicht werden, nämlich die Einschränkung, dass alle Fälle abgedeckt werden müssen. Mit einem Sealed Interfaces kann demnach auch der default-Case entfernt werden, falls alle Möglichkeiten im Switch behandelt werden.

Gerade dieser default-Case ist mit dem Visitor-Pattern auch schwer zu implementieren. Es müssten alle nicht implementierten Methoden mit default auf eine weitere Methode verweisen, welche den allgemeinsten Fall abdeckt.

```
1  sealed interface Node permits Value, Add {}
2
3  record Value(int value) implements Node {}
4  record Add(Node left, Node right) implements Node {}
5
6  static int eval(Node node) {
7      return switch (node) {
8          case Value v -> v.value;
9          case Add a -> eval(a.left) + eval(a.right);
10     };
11 }
```

Listing 12: AST-Evaluierer umgesetzt in modernem Java

Der Visitor wird komplett durch eine statische Methode ersetzt, die rekursiv aufgerufen wird. Node wurde durch ein Sealed Interface ersetzt, welches die beiden Record-Typen Value und Add erlaubt. Die Methode `accept` wird ebenfalls nicht mehr benötigt.

Zusammengefasst kann gesagt werden, dass gegenüber dem Visitor-Pattern viel Code eingespart werden kann. Die Möglichkeit der Vererbung bei verschiedenen konkreten Visitor-Klassen ist dazu im Vergleich eher ein weniger gebrauchtes Feature.

5.2 Zusammenfassung und Ausblick

Im Laufe dieser Arbeit wurde der Codegenerierer für JavaTX neu geschrieben. Dabei wurde ein neuer AST implementiert. Durch einen Verarbeitungsschritt worde der AST des Parsers in diesen übersetzt. Zu Hilfe genommen wurden Rekordtypen, sealed Interfaces und der in Java 18 neu eingeführte Typswitch. Dadurch wurde die Codequalität im Vergleich zu dem bereits vorhandenen Bytecodegenerierer verbessert. Die Implementierung ist modular aufgebaut und erlaubt in Zukunft ein einfaches Anpassen.

Nicht implementiert wurden Optimierungen, wie das direkte Verwenden von `if_icmp` in If-Statements oder das Löschen von nicht benutztem Code. Der AST wird sehr wörtlich übernommen. Hier könnte in Zukunft noch eine bessere Implementierung verwendet werden.

Als ein Zusatzschritt vor dem Codegenerierer wurden die Typinformationen des Typunifiers in den neuen AST eingesetzt. Die übrig gebliebenen Constraints wurden als Java Generics auf die Klassen und Methoden verteilt. Wildcardtypen funktionieren momentan noch nicht, hier ist also noch eine Anpassung nötig.

Ebenfalls fehlt die Implementierung der überladbaren Funktionstypen, aber diese sollten kaum Anpassung am Code nötig machen.

Literatur

[1] Gavin Bierman. *JEP 397: Sealed Classes (Second Preview)*. 11. März 2022. URL: https://openjdk.org/jeps/397.

[2] Gavin Bierman. *JEP 427: Pattern Matching for switch (Third Preview)*. 18. Juli 2022. URL: https://openjdk.org/jeps/427.

[3] Luis Damas und Robin Milner. „Principal Type-Schemes for Functional Programs“. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, S. 207–212. ISBN: 0897910656. DOI: 10.1145/582153.582176.

[4] Ian Darwin. *The Visitor Design Pattern in Depth*. 15. Juli 2019. URL: https://blogs.oracle.com/javamagazine/post/the-visitor-design-pattern-in-depth.

[5] Brian Goetz. *JEP 384: Records (Second Preview)*. 11. März 2022. URL: https://openjdk.org/jeps/384.

[6] 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*. ACM, 2017, S. 248–256. ISBN: 978-1-4503-5340-3. DOI: 10.1145/3132190.3132203.

[7] Martin Plümicke und Florian Steurer. „Erweiterung und Neuimplementierung der Java Typunifikation“. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Faculty of Mathematics und Natural Sciences, University of Oslo, 2018, S. 134–149. ISBN: 978-82-7368-447-9.

[8] Martin Plümicke und Etienne Zink. *Java-TX: The language*. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2022. DHBW Stuttgart, 2022. URL: https://www.dhbw-stuttgart.de/fileadmin/dateien/Forschung/Forschungsschwerpunkte_Technik/DHBW_Stuttgart_INSIGHTS_1_2022_Java-TX_The_language.pdf.

[9] James C. Tiernan. „An Efficient Search Algorithm to Find the Elementary Circuits of a Graph“. In: *Commun. ACM* 13.12 (Dez. 1970), S. 722–726. ISSN: 0001-0782. DOI: 10.1145/362814.362819. URL: https://doi.org/10.1145/362814.362819.

[10] Jim Trevor. „What are principal typings and what are they good for?“ In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*. 1996, S. 42–53. DOI: 10.1145/237721.237728.