Heterogene Übersetzung echter Funktionstypen

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart, Campus Horb

von

Etienne Zink

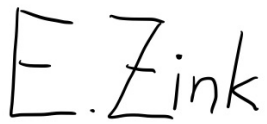
30.05.2022

Bearbeitungszeitraum	300 Stunden
Matrikelnummer, Kurs	7213853, HOR-TINF2019
Ausbildungsfirma	fischerwerke GmbH & Co. KG, Waldachtal
Betreuer	Prof. Dr. Martin Plümicke
Gutachter	Prof. Dr. Martin Plümicke

Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung der Ausbildungsstätte vorliegt.

Horb, 30.05.2022

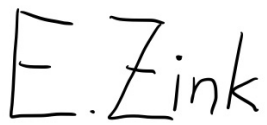


Etienne Zink

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: *Heterogene Übersetzung echter Funktionstypen* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Horb, 30.05.2022



Etienne Zink

Zusammenfassung

Java bietet seit Version 5 Unterstützung für generische Parameter an. Die Übersetzung derer erfolgt seit jeher homogen. Dies bedeutet, dass jede Permutation eines Typs in dessen generischen Parameter auf den gleichen Typ abgebildet wird. Eine solche Abbildung wird seit dem als *Type Erasure* bezeichnet.

Java-TX führt einige neue Funktionalitäten in Java ein. Darunter zählen unter anderem die globale Typinferenz und Unifikation. Außerdem wurden analog zu Scala echte Funktionstypen eingeführt. In Java-TX werden diese durch ein Interface `Fun`N`$$` dargestellt. Dieses Interface beinhaltet dabei generische Parameter für die Argumenttypen und den Rückgabetyp einer Funktion. Unter Berücksichtigung der Type Erasure werden die Funktionstypen homogen Übersetzt.

Diese Arbeit stellt einen Ansatz vor, um die in Java-TX eingeführten echten Funktionstypen heterogen zu übersetzen. Grundlage für diese Übersetzung ist dabei der Ansatz aus Pizza [OW97]. Für jeden Funktionstyp wird ein Interface erstellt, welches das Basis Interface `Fun`N`$$` spezialisiert. Dabei unterscheiden sich die einzelnen Spezialisierungen in deren Namen, welche sich unter anderem aus den Deskriptoren der generischen Parametern zusammensetzen. Der in dieser Arbeit vorgestellte Ansatz wurde anschließend in Java-TX implementiert.

Abstract

Java offers support for generic parameters since version 5. The translation of these has always been homogeneous. This means that every permutation of a type in its generic parameter is mapped to the same type. Such a mapping is called *Type Erasure* since then.

Java-TX introduces several new functionalities to Java. Among them are global type inference and unification. In addition, real function types have been introduced analogous to Scala. In Java-TX these are represented by an interface `Fun`N`$$`. This interface contains generic parameters for the argument types and the return type of a function. Considering the type erasure, the function types are translated homogeneously.

This work presents an approach to heterogeneously translate the real function types introduced in Java-TX . The basis for this translation is the approach from Pizza [OW97]. For each function type, an interface is created which specializes the base interface `Fun`N`$$`. Here, the individual specializations differ in their names, which are composed, among other things, of the descriptors of the generic parameters. The approach presented in this paper was subsequently implemented in Java-TX .

Inhaltsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface
AST Abstract Syntax Tree
CLI Common Language Infrastructure
CLR Common Language Runtime
LINQ Language Integrated Query
TPH Typ-Platzhalter
JIT Just In Time Compiler
JVM Java Virtual Machine

Abbildungsverzeichnis

Quellcodeverzeichnis

1. Einführung

Dieses Kapitel beinhaltet eine allgemeine Einführung in diese Arbeit. Dabei wird zuerst eine kurze Einführung in das Projekt Java-TX gegeben. Anschließend erfolgt die Erläuterung der Motivation dieser Arbeit. Somit wird die Problemstellung und die daraus resultierende wissenschaftliche Fragestellung näher betrachtet. Basierend auf dieser Problemstellung wird das Ziel dieser Arbeit näher definiert. Das Ziel der Arbeit wird dabei in zwei Kategorien untergliedert. Es bestehen sowohl theoretische, als auch praktische Ziele. In diesem Kapitel wird darauf verzichtet die genaue Vorgehensweise und den Aufbau der Arbeit näher zu erläutern. Beide Themen können im Kapitel 6 „Zusammenfassung“ nachgelesen werden.

1.1. Java-TX

Nach [Plü18] handelt es sich bei Java-TX (stehend für Java **T**ype e**X**tended) um eine Weiterentwicklung der Sprache Java. Dabei umfasst Java-TX unter anderem einen Compiler für diese Sprache. Außerdem beinhaltet das Projekt Java-TX ein Plugin für die Entwicklungsumgebung Eclipse. Dieses Plugin ermöglicht es, eine Kompilation mit Typinferenz direkt in Eclipse durchzuführen. Dies wird durch das in dem Plugin enthaltene `jar`-Archiv erreicht. Des Weiteren kann das Plugin den vom Compiler erstellten Parsebaum grafisch darstellen. Errechnete Typen für die verwendeten Bezeichner werden dabei als Tooltips dargestellt.

Der Compiler in dem Projekt Java-TX ist eine unabhängige Komponente, laut [Plü18]. Dieser basiert auf der Java Version 8. Ausschlaggebend für diesen Compiler sind jedoch die hinzugefügten Funktionalitäten. So weist dieser eine globale Typinferenz, echte Funktionstypen, prinzipale Typen aber auch Durchschnittstypen auf. Zum Verständnis dieser Arbeit sind ausschließlich die ersten beiden Neuerungen von größerem Interesse. Daher werden diese im Abschnitt 2.4 „Neue Funktionalitäten in Java-TX “ näher erläutert. Bei globaler Typinferenz handelt es sich in gekürzter Fassung darum, die Typdeklarationen im Quellcode weglassen zu können. Dabei liegt die Besonderheit darin, dass dennoch der Quellcode statisch typisiert ist. Echte Funktionstypen werden dazu benutzt, Lambda-Ausdrücken in Java einen konkreten Typ zuzuordnen. Diese Funktionalitäten spiegeln sich nicht nur theoretisch, sondern auch in einer entsprechenden Implementierung wieder. So wurde im Rahmen mehrerer studentischer Arbeiten ein Prototyp für diesen Compiler implementiert.

Mithilfe dessen kann Java Quellcode, welcher der Version 8 und den erwähnten Funktionalitäten entspricht, in Bytecode übersetzt werden. Dieser Bytecode kann dabei auf unterschiedlichen Java Virtual Machines (JVMs) ausgeführt werden. Die entsprechende JVM muss hierfür Bytecode der Version 1.5 unterstützen.

1.2. Motivation

Die Motivation dieser Arbeit ist durch die bestehenden Problemen in der Typinferenz und Typunifikation in Kombination mit Javas Type Erasure begründet. Die Typunifikation berechnet nach der Typinferenz den prinzipalen Typen für die dabei eingesetzten Typ-Platzhalters (TPHs). Somit ist der Quellcode statisch typisiert. Unter anderem können dabei auch Funktionstypen inferiert werden. Auf diese Funktionstypen wird genauer im Abschnitt 2.4 „Neue Funktionalitäten in Java-TX “ eingegangen. Die in Java-TX eingeführten echten Funktionstypen besitzen generische Parameter. Diese geben Auskunft über die Typen der Attribute und der Rückgabe der Methode `apply`. Die Methode `apply` stellt die einzige Methode in einem Funktionstypen dar. Wird der Code in Listing 2.2 betrachtet, so können zwei verschiedene Funktionstypen als formale Parameter inferiert werden. Diese Funktionstypen unterschieden sich ausschließlich in deren Typ-Parametern. Der Intersektionstyp der Methode `m` ist in Abbildung 2.3 dargestellt. Nach der sogenannten *Type Erasure* von Java, können die formalen Parameter nicht mehr unterschieden werden. So ergibt sich der Intersektionstyp aus Abbildung 2.4 für `m`. Aufgrund dessen, handelt es sich bei der Methode `m`, nach der Type Erasure, um eine Methode, deren Implementierungen nicht unterschieden werden können. Somit ist die Methode `m` in Java nicht zulässig. Eine genauere Beschreibung dieses Problems ist im Abschnitt 2.2 „Aktuelle Übersetzung generischer Parameter“ zu finden. In diesem wird außerdem die Type Erasure genauer erläutert. Das beschriebene Problem ist eine Auswirkung der Type Erasure. Weitere Auswirkungen dieser werden im Abschnitt 2.2 „Aktuelle Übersetzung generischer Parameter“ beschrieben.

Somit muss das Problem der Type Erasure in Java-TX gelöst werden, sodass die Typunifikation korrekt durchgeführt werden kann. Mit der Type Erasure müsste eine Logik in Java-TX implementiert werden, welche derartige Unifikation vorbeugt oder löscht. Dies würde aber dem angestrebten Ziel, generelleren Code durch die Typunifikation zu generieren, widersprechen. Daher muss eine theoretische Lösung gefunden und implementiert werden, sodass genereller Code in Java-TX compiliert werden kann. Spätere Software-Entwickelnde profitieren durch generellen Code, indem Methoden vielseitiger einsetzbar sind. Dies könnte die Weiterentwicklung durch einfachere Erweiterbarkeit verbessern. Eine weitere Motivation liegt in der Erweiterbarkeit des Ansatzes auf beliebige Klassen, welche

generischen Parametern besitzen. Die Evaluation dessen, kann jedoch nicht im Rahmen dieser Arbeit erfolgen.

1.3. Ziel der Arbeit

Das Ziel dieser Arbeit ist, die Problematik der Type Erasure in Java-TX für echte Funktionstypen zu lösen. Somit soll es zukünftig möglich sein, dass Klassen wie in Listing 2.2 compiliert und auf der JVM ausgeführt werden können. Dadurch entsteht die folgende wissenschaftliche Fragestellung für diese Arbeit:

Wie können echte Funktionstypen, mit gleicher Parameteranzahl, in Java-TX , trotz Type Erasure, unterschieden werden?

Diese Fragestellung muss in mehreren Teilschritten beantwortet werden. So ist zuerst zu beantworten, wie die aktuelle Übersetzung generischer Parameter, die sogenannte Type Erasure, funktioniert. In diesem Zuge muss betrachtet werden, weshalb diese Übersetzung im Design von Java bzw. der JVM gewählt wurde. Anschließend muss dargestellt werden, wie sich diese Übersetzung auf den Bytecode, genauer auf die Signaturen und Deskriptoren der Typen mit generischen Parametern, auswirkt. Im Zuge dessen müssen die mit der Type Erasure einhergehenden Probleme näher erläutert werden.

Nachdem die Problematik der Type Erasure erläutert wurde, müssen Ansätze für eine Lösung des Problems gefunden werden. Da bereits Ansätze zur Lösung des Problems existieren (siehe Abschnitt 2.3 „Übersetzung generischer Parameter in Java vergleichbaren Sprachen“), müssen diese dargestellt werden. Dies entspricht der abgeleiteten Frage: Welche Ansätze zur Lösung der Type Erasure existieren bereits? Um diese Frage zu Genüge zu beantworten, sollten auch mögliche Ansätze anderer Sprachen betrachtet werden. Außerdem impliziert diese Frage eine Evaluation der gefundenen Ansätze. Daher muss anschließend beantwortet werden: Welche Vor- und Nachteile bergen die Ansätze zur Lösung der Type Erasure? Durch die Beantwortung dieser Frage, werden die Lösungen vergleichbar. Aufgrund der Größe der Evaluation, muss auf diese im Rahmen dieser Arbeit verzichtet werden.

Zum Transfer der gefundenen Lösungen ist es unabdingbar, dass die neu eingeführten Funktionalitäten in Java-TX erläutert werden. So muss beantwortet werden: Was sind echte Funktionstypen? Im diesem Zuge ist außerdem zu klären: Worum handelt es sich bei der Typinferenz und Typunifikation in Java-TX? Erst durch Beantwortung dieser Fragen, können die Lösungsansätze im Kontext von Java-TX diskutiert werden.

Anschließend muss die Frage gestellt werden: Welche Lösungen für die Type Erasure existieren für Java-TX? Somit müssen theoretische Lösungen gegeben werden. Darauf aufbauend muss beantwortet werden: Welche Lösung bietet sich am Besten für Java-TX an? Dabei müssen diverse Aspekte, wie die Speicherplatz- und die Laufzeitveränderung, betrachtet werden. Basierend auf den zu betrachtenden Aspekten, muss somit die passendste Lösung ausgewählt werden. Als sogenannter *Proof of Concept* muss die entsprechende Lösung in Java-TX implementiert werden. Diese Implementierung folgt der Frage: Wie kann der gewählte Ansatz in Java-TX umgesetzt werden? Zur Beantwortung dieser Frage ist es von Nöten, die implementierte Funktionalität anschließend auf deren Korrektheit zu prüfen. Daher ist die abschließende Frage zu klären: Wurde die gewählte Lösung korrekt in Java-TX umgesetzt?

Auf Grundlage der diversen, zu beantwortenden Fragen, müssen unterschiedliche Artefakte erstellt werden. So untergliedern sich die Artefakte in deren theoretische und praktische Natur. Theoretischer Natur sind die Recherchen, welche zur Type Erasure, Java-TX und den bereits existierenden Lösungen durchgeführt werden müssen. Somit entstehen aus diesen Fragen Informationssammlungen als Artefakte. Als potentielle Lösungen für die Type Erasure müssen Konzepte als Artefakte, erstellt werden. Diese umfassen dabei sowohl eine Beschreibung dieser, als auch deren Vor- und Nachteile. Praktischer Natur sind anschließend die Artefakte des Quellcodes, zur Implementierung und Verifikation der Lösung. Darunter fällt auch der Quellcode, welcher für die Verifikation der Umsetzung benötigt wird. Dieser stellt dabei einen Prototyp dar.

2. Generische Parameter

2.1. Generische Parameter in Java

Umsetzung generischer Parameter in Java

Laut [Ora21b] gilt eine Klasse oder Methode als *generisch*, wenn diese mindestens eine Typvariable deklariert. Eine solche Typvariable wird in diesem Zusammenhang als *Typpparameter* der Klasse bezeichnet. Der Typparameter-Abschnitt einer Klasse folgt im Anschluss auf deren Namensdeklaration. Hierbei werden diese zwischen spitzen Klammern, Komma separiert deklariert. Bei einer Typvariable handelt es sich um einen unqualifizierten Bezeichner. Dieser wird als Typ in einem Interface, einem Konstruktor-Körper, einer Klasse oder einer Methode verwendet. Eine solche Typvariable wird durch die Deklaration eines Typparameters einer generischen Klasse, Methode, Konstruktor oder Interfaces eingeführt.

Als parametrisierter Typ wird laut [Ora21b] eine Klasse oder Interface bezeichnet, welche/s generisch ist. Ein Beispiel eines solchen parametrisierten Typen wäre die Klasse `C<`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`>`. Der Bezeichner `C` stellt dabei den Namen der Klasse oder des Interfaces dar. Bei der darauf folgenden Sektion `<`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`>`, handelt es sich um eine List aus Typargumente. Diese Liste aus Typargumenten deutet dabei auf eine bestimmte Parametrisierung der Klasse oder des Interfaces hin. Somit handelt es sich dabei um eine bestimmte Parametrisierung, welche die Typen $\mathtt{T}_i$ umfasst. Typargumente und Typparameter stehen somit im gleicher Korrelation zueinander, wie klassische Argumente und Parameter. Wenn eine Klasse oder ein Interface `A<`$\mathtt{F}_1$`,...,`$\mathtt{F}_n$`>` deklariert wird, so existiert nach [Ora21b] zu jedem Typparameter $\mathtt{F}_i$ eine entsprechende Grenze $\mathtt{B}_i$. Die Typargumente $\mathtt{T}_i$ erstrecken sich dabei über alle Typen und Subtypen der entsprechende Grenze $\mathtt{B}_i$.

Die Menge der möglichen Parametrisierungen der Klasse `A` könnte somit wie folgt definiert werden: $\{\mathtt{A<T}_1\mathtt{,...,T}_n\mathtt{>} \mid \mathrm{T}_i <: \mathrm{B}_i,\ 1 \leq i \leq n\ \}$.

Nach [Ora21b] können Typargumente entweder Referenztypen oder sogenannte *Wildcards* sein. Referenztypen sind: Klassen-Typen, Interface-Typen, Typvariablen und Array-Typen. Somit handelt es sich bei den Referenztypen um eine disjunkte Menge gegenüber den primitiven Typen in Java. Wildcards werden verwendet, wenn ausschließlich begrenzte Informationen der Typen benötigt werden. Die Grammatik für eine Wildcard ist in

[Ora21b] wie folgt definiert: `{Annotation} ? [WildcardBounds]`. Bei der *Annotation* handelt es sich um übliche Annotationen aus Java. Die *WildcardBounds* können ausgelassen werden, dann spricht man von einer *ungebundenen Wildcard*. Werden die *WildcardBounds* verwendet, so wird die Wildcard als *gebundene Wildcard* bezeichnet. Es existieren dabei zwei verschiedene Arten von gebundenen Wildcards. So können Wildcards mit einer Grenze `B` wie folgt deklariert werden:

`? extends B` deklariert eine Wildcard mit der oberen Grenze `B` (*Extends-Wildcard*).

`? super B` deklariert eine Wildcard mit einer unteren Grenze `B` (*Super-Wildcard*).

Nach [Ora21b] basieren Wildcards auf der Arbeit von [IV02]. Ähnlich zu Wildcards, können auch Typparameter als gebundene Typparameter formuliert werden. Dies geschieht analog mit den Schlüsselwörtern `extends` und `super`.

Einführung generischer Parameter in Java

Generische Parameter wurden nach [GH05] in Java 5 eingeführt. Dabei wird betont, dass generische Parameter die Typsicherheit erhöhen. Dies liegt unter anderem daran, dass durch generische Parameter sichergestellt werden kann, dass Objekte einem bestimmten Typ genügen. Außerdem können hierdurch allgemeinere Klassen erstellt werden. So existiert die Klasse `Pair<A, B>`, welche ein Paar aus beliebigen Typen darstellt. Hierdurch ist es nicht nötig, eine Klasse für jede zweistellige Typkombination zu erstellen. Des Weiteren können durch generische Parameter Subklassen dazu gezwungen werden, passende Implementierungen umzusetzen. Ein Beispiel hierzu ist in [GH05] bei der Klasse `DBFactory<T extends DBPeer>` dargestellt. Somit definieren generische Parameter Constraints für die Spezialisierungen oder Implementierungen und deren Methoden dar. In [GH05] werden für generische Parameter in Kombination mit gebundenen Wildcards zwei weitere Vorteile erläutert. So können Wildcards zur Entkopplung einer Schnittstelle und der Implementierung beitragen. Dies kann durch eine obere Schranke (Extends-Wildcard) in einem Interface bzw. der Methodensignatur erfolgen. Durch die Verwendung einer oberen Schranke als Rückgabetyp, kann die Implementierung eine Spezialisierung dieser Schranke intern verwenden und zurückgeben. Bei der Verwendung dieser Methode wird dieses Interna transparent durch die Schranke und ausschließlich diese kann als Typ angenommen werden. Ein weiterer Vorteil ist die Verwendung von Wildcards in der Methodensignatur einer (abstrakten) Methode. So führt dies dazu, dass die Signatur einer Implementierung dieser Methode von deren ursprünglichen Signatur abweichen kann. Beispielhaft wäre hierfür die Klasse `NumberGenerator` in [GH05]. Hierbei verwendet das Interface eine obere Schranke in deren Signatur. Aufgrund dessen kann die Signatur der Spezialisierung (`FibonacciGenerator`) angepasst werden. Zu beachten ist, dass die neue Signatur dennoch

den Constraints der ursprünglichen genügen muss. So handelt es sich in diesem Beispiel bei einer `List<Integer>` noch um eine `List<? extends Number>`. Die Signatur wurde in diesem Fall spezifischeren bzw. die Methode der Klasse `FibonacciGenerator` liefert einen spezifischeren Typ zurück. Dennoch wird dieser spezifischere Typ den Programmierenden, durch die Wildcard, transparent dargestellt.

In [GH05] wurde außerdem die große Anwendung von generischen Parametern in der Klasse `java.util.Collections` der Standardbibliothek erwähnt. Diese wurde an einigen Beispiele dieser Klasse demonstriert. Außerdem wurden darin bereits erste Probleme der Übersetzung generischer Parameter erwähnt. Dies wird am Beispiel der versuchten Instanziierung eines generischen Arrays näher erläutert. Es ist nicht möglich, direkt eine Instanz eines Arrays zu erstellen, welches dem generischen Parameter entspricht. Eine Erklärung dazu findet sich in der Implementierung von generischen Parametern, der sogenannten *Type Erasure* (siehe Abschnitt 2.2 „Aktuelle Übersetzung generischer Parameter“).

2.2. Aktuelle Übersetzung generischer Parameter

Vorbedingungen an die Übersetzung

Bei der aktuellen Übersetzung von generischen Parametern spielt der Leitgedanke von Java eine bedeutende Rolle. Die Designprinzipien hinter Java waren laut [Ora22]:

- Architekturneutralität
- Portabilität
- Dynamische Anpassbarkeit

Durch diese soll Java nach [Ora22] die Anforderungen erfüllen sichere, hoch performante und robuste Software für multiple Plattformen in heterogenen und verteilten Netzwerken zu entwickeln. Um die Anforderung der Portabilität umzusetzen, wird der Java-Sourcecode im Gegensatz zu C nicht direkt in Maschinencode, sondern in *Bytecode* übersetzt. Bytecode stellt ein Zwischenprodukt von Code dar, welcher vom Java Compiler generiert wurde. Dieser Bytecode ist Architektur und Software neutral. Somit wird der selbe Bytecode auf unterschiedlichen Systeme auf Basis des gleichen Sourcecodes erstellt. Diese Systeme können sich dabei sowohl in der Hardware, als auch in der Software bzw. dem Betriebssystem unterscheiden. Bei der Ausführung wurde zuerst der Bytecode interpretiert [Ora22] und anschließend durch einen Just In Time Compiler (JIT) [Ora19] zur Laufzeit ausgeführt. Hierdurch konnte das Problem der Code Varianten gelöst werden. Einmal kompilierter

Sourcecode von Java verfügt somit als Bytecode über die Eigenschaft, auf jeder Plattform ausführbar zu sein. Eine weitere Komponente um dieses Ziel zu erreichen, stellt die architekturneutrale und portable Laufzeitumgebung von Java dar. Dabei handelt es sich laut [Ora22] um die sogenannte *Java Virtual Machine (JVM)*. Diese stellt eine Spezifikation einer abstrakten Maschine für Java Bytecode dar, welcher auf dieser ausführbar ist. In dieser Spezifikation ist definiert, dass die JVM selbst nicht die Architekturneutralität erfüllen muss.

Wie zu Beginn von Kapitel 2 „Generische Parameter“ erläutert, wurden generische Parameter in Java 5 eingeführt. Somit existierte die Spezifikation der JVM vor diesen. Aufgrund dessen existierte auch die Definition des Bytecodes vor den generischen Parametern. Bei der Einführung einer neuen Funktionalität muss somit entweder der Bytecode angepasst werden, oder eine weitere Umsetzung mithilfe der bestehenden Bytecode-Definition entwickelt werden. Die Bytecode-Definition kann nach den Designprinzipien von Java jedoch nur begrenzt angepasst werden. Ansonsten würde dies dazu führen, dass bereits existierender Bytecode nicht mehr ausführbar ist. Dies würde damit zu einer Verletzung der Abwärtskompatibilität führen und Legacy Code müsste möglicherweise angepasst oder neu kompiliert werden. Aus diesem Grund wurde zur Einführung der generischen Parameter die sogenannte *Type Erasure* eingeführt.

Type Erasure

Bei der Type Erasure handelt es sich nach [Ora21b] um eine Abbildung von Typen. Somit bildet die Type Erasure Typen, welche möglicherweise parametrisierte Typen oder Typvariablen besitzen, auf Typen ab, welche keine parametrisierten Typen oder Typvariablen besitzen. Die generischen Parameter eines Typen werden somit „gelöscht (eng. *erased*)“. Folgend wird wie in [Ora21b] ein Typ `T` nach der Type Erasure als `|T|` geschrieben. Der einstellige Operator `|`X`|` mit dem Operand X kann somit als *Erasure-Operator* betrachtet werden. Ein parametrisierte Typ `T<A, B>` wird somit auf den Typ `|T<A, B>|` = `|T|` abgebildet.

Folgende Abbildungen wurden in [Ora21b] für die Type Erasure definiert:

1. `|A<`T_1`, ..., `T_n`>|` := `|A|`,
2. `|A.B|` := `|A|.B`,
3. `|A[]|` := `|A|[]`,
4. `|`*Typvariable T*`|` := `|`*Linkeste Grenze von T*`|`,
5. `|T|` := `T` sonst.

Um eine korrekte Auflösung der Typen nach der Type Erasure zu gewährleisten, müssen Signaturen entsprechend angepasst werden. Deshalb beinhaltet die Type Erasure auch die Abbildung von Signaturen. So definiert [Ora21b] das eine Signatur, welche möglicherweise parametrisierte Typen oder Typvariablen besitzt, auf eine Signatur ohne diese abgebildet wird. Dabei kann es sich um Signaturen von sowohl Konstruktoren, als auch Methoden handeln. Es existiere eine Signatur s, so wird $|s|$ definiert als eine Signatur mit demselben Namen wie s und der Type Erasure aller Typen der formalen Parameter in s. Außerdem wird die Type Erasure auf den Rückgabetyp und die Typparameter einer generischen Methode oder Konstruktor angewandt, falls deren Signatur *erased* wird. Die Signatur einer generischen Methode enthält somit nach der Type Erasure keine Typparameter mehr. Dies führt dazu, dass nach Abschluss der Type Erasure die Signaturen zu den Typen kompatibel sind. Somit handelt es sich bei der Type Erasure um eine homogene Übersetzung generischer Parameter.

Ein Beispiel für die Type Erasure in Java-TX ist in Listing 2.1 dargestellt. Die Klasse `OLFun` verfügt darin über die Methode `m`. Typannotationen konnten durch Java-TX ausgelassen werden (siehe Abschnitt 2.4 „Neue Funktionalitäten in Java-TX “). Außerdem werden in dieser Arbeit die `import` Anweisungen der Übersichtlichkeit halber vernachlässigt. Die Funktionalität der Methode `m` besteht darin, den Parameter `f`, welcher eine Funktion darstellt, auf den formalen Parameter `x` anzuwenden. Dabei wird diese nicht direkt auf `x`, sondern auf `x+x` angewandt. Somit muss `x` ein Typ besitzen, für welchen der Operator `+` definiert ist. Die Funktion `f` muss dabei wiederum den Typ von `x` als Typ des formalen Parameter der Methode `apply` besitzen. Der Rückgabetyp von `m` entspricht erneut dem Typen von `x`, da dieser dem formalen Parameter zugewiesen wird. Eine genauere Erläuterung der Typinferenz eines solchen Codes wird in Abschnitt 2.4 näher erläutert.

```
class OLFun {
    m(f, x){
        x = f.apply(x+x);
        return x;
    }
}
```

Listing 2.1: Klasse `OLFun` in Java-TX mit zusätzlichem Parameter `x`.

Nun ist es nötig, den Typ der Methode `m` zu bestimmen. Dieser wird für die zu erstellende Signatur im Bytecode benötigt. Für die Methode `m` ergibt sich der Intersektionstyp aus Abbildung 2.1.

`Fun1$$<Double, Double>` × `Double` → `Double` &
`Fun1$$<Integer, Integer>` × `Integer` → `Integer`

Abbildung 2.1.: Intersektionstyp von Listing 2.1.

Dieser Intersektionstyp entspricht dem korrekten Typ, und kann in den entsprechend Java Bytecode kompiliert werden. Durch die aktuelle Übersetzung generischer Parameter, in diesem Fall der Typargumente von `Fun1$$`, entsteht der Intersektionstyp aus Abbildung 2.2. Dies kann durch den (korrekt Typ annotierten) Quellcode der Klasse `OLFun` verifiziert werden. Der Quellcode ist in Listing L.1 dargestellt. Hierbei werden beide Ausprägungen der Methode `m` als separate Implementierung angegeben. Des Weiteren werden die Signaturen der Methode `m` in Abbildung A.1 und die Deskriptoren in Abbildung A.2 dargestellt.

Die „Subtypen“ des Intersektionstyp weisen somit keinen Unterschied im Typ von `f` auf. Die Funktion `f` besitzt jeweils den Typ `Fun1$$`. Dennoch kann Java-TX die Klasse `OLFun` aus Listing 2.1 und Java die selbe Klasse in Listing L.1 kompilieren. Zur Unterscheidung der Implementierungen von `m` genügt diesen der Unterschied der Typen im zweiten formalen Parameter der Signatur.

$$\texttt{Fun1\$\$} \times \texttt{Double} \to \texttt{Double \&}$$
$$\texttt{Fun1\$\$} \times \texttt{Integer} \to \texttt{Integer}$$

Abbildung 2.2.: Intersektionstyp von Listing 2.1 nach der Type Erasure.

Problematiken durch die Type Erasure

Nach [Ora21b] besitzen zwei Methoden oder Konstruktoren die selbe Signatur, wenn beide die gleichen Namen, Typparameter und formalen Parameter besitzen. Sollten sich formale Parameter ausschließlich in der Reihenfolge unterscheiden, so gelten diese Signaturen nicht als die selben. Eine Signatur einer Methode gilt dabei nach [Ora21b] als *Subsignatur*, wenn diese die selbe Signatur wie eine weitere Methode aufweist. Dies gilt auch nach der Anwendung der Type Erasure auf eine Signatur. Somit gelten zwei Signaturen, welche sich ausschließlich in den Namen der Typvariablen unterscheiden, als die selben. [Ora21b] definiert zwei Signaturen als *überschreibungs Äquivalent*, sollte eine der beiden Signaturen eine Subsignatur der anderen sein. Existieren zwei Methoden oder Konstruktoren in einer Klasse, welche überschreibungs äquivalente Signaturen besitzen, so wirft Java einen Kompilierfehler. Ausgelöst wird dieser auch bei abstrakten Methoden.

Das keine überschreibungs äquivalenten Methoden oder Konstruktoren definiert werden können, liegt der Bestimmung der Methodensignatur zu Laufzeit zugrunde [Ora21b]. Bei der Bestimmung der aufzurufenden Methoden zur Laufzeit, wird die passende Methode anhand deren Name und Argumente bestimmt. Zuerst wird dafür eine Menge möglicher Methoden aufgestellt, welche sowohl aufrufbar, als auch angemessen sind. Eine solche Methode wird als *korrekt aufrufbar basierend auf den gegebenen Argumenten* bezeichnet. Durch die Definition einer Menge möglicher Methoden, muss der Java Compiler die passendste Methode bestimmen. Die am Methode, welche am spezifischsten ist, wird dabei

als die am passendsten definiert. Bei der Bestimmung der möglichen Methoden kann in drei Phasen geschehen. Jede Phase versucht angemessene Methoden zu bestimmen. Sollte in einer Phase keine solche Methode gefunden werden, so wird die nächste Phase begonnen. Wenn am Ende der dritten Phase keine angemessene Methode bestimmt werden konnte, schlägt die Kompilierung fehl. Die einzelnen Phasen können [Ora21b] entnommen werden. Nachdem die am spezifischsten Methode bestimmt wurde, kann zur Laufzeit deren Deskriptor für den Methodenaufruf verwendet werden. Der Deskriptor setzt sich dabei laut [Ora21b] aus der Signatur und dem Rückgabetyp der Methode zusammen. Aus diesem Grund, können keine Methoden überladen werden, welche sich ausschließlich in den Namen von Typvariablen unterscheiden. Die Überladung zweier Methoden ist ausschließlich gestattet, sollten sich deren Signatur unterscheiden [Ora21b]. Dies würde ansonsten dazu führen, dass nach der Type Erasure gleiche Deskriptoren, für unterschiedliche Methoden existieren. Somit könnte die JVM nicht feststellen, welche Methode aufgerufen werden muss.

Ein Beispiel dieses Umstandes ist in Listing 2.2 dargestellt. Dabei handelt es sich um die Klasse `OLFun` in Java-TX , welche die Methode `m` besitzt. Im Unterschied zu Listing 2.1, weist die Signatur von `m` nur einen formalen Parameter auf. Bei diesem handelt es sich erneut um eine Funktion vom Typ `Fun1$$`. Die Variable `x` ist eine lokale Variable.

```
class OLFun {
    m(f){
        var x = 1;
        x = f.apply(x+x);
        return x;
    }
}
```

Listing 2.2: Klasse `OLFun` in Java-TX ohne zusätzlichen Parameter `x`.

Der Intersektionstyp der Methode `m` ist dabei in Abbildung 2.3 dargestellt.

```
Fun1$$<Double, Double> → Double &
Fun1$$<Integer, Integer> → Integer
```

Abbildung 2.3.: Intersektionstyp von Listing 2.2.

Dieser Intersektionstyp wird bei der Übersetzung in Java Bytecode der Type Erasure unterzogen. Somit ändert sich dieser zum Typ aus Abbildung 2.4.

```
Fun1$$ → Double &
Fun1$$ → Integer
```

Abbildung 2.4.: Intersektionstyp von Listing 2.2 nach der Type Erasure.

Wie zu erkennen ist, unterscheiden sich die „Subtypen“ des Intersektionstyp ausschließlich in deren Rückgabetyp. Da nach [Ora21b] eine solche Überladung nicht gestattet ist, kann diese Klasse nicht kompiliert werden. Somit ist ein theoretisch korrekter Typ, durch die aktuelle Übersetzung generischer Parameter, nicht zulässig. Dieser Umstand führt dazu, dass das angestrebt Ziel von Java-TX , generellen Code zu erstellen, an der Klasse `OLFun` aus Listing 2.2 scheitert.

Zudem existieren weitere Problematiken, welche in Java mit der Type Erasure einhergehen. Diese sind unter anderem die folgenden:

`instanceof` Operator wird verwendet, um die den Typ einer Variablen abzufragen. So evaluiert dieser nach [Ora21b] zu wahr (`true`), wenn die Referenz ungleich `null` ist und diese ohne `ClassCastException` zu dem Referenztyp umgewandelt werden kann. Ansonsten evaluiert dieser zu unwahr (`false`). Die logische Konsequenz der Type Erasure lässt folgen, dass dieser nicht die Typargumente berücksichtigen kann. Somit geschieht die Evaluierung ausschließlich auf Basis des Referenztyp, als würde dieser keine Typparameter besitzen. Dies hatte nach [Ora20] zur Folge, dass Typen mit Typparameter nicht als Referenztyp für den `instanceof` Operator verwendet werden durften. Seit Java 16 und der Einführung des Pattern Matchings durch den `instanceof` Operator, dürfen Referenztypen mit Parametern für diesen verwendet werden [Ora21a]. Dennoch kann dieser keine Unterscheidung der Typargumente leisten, da die Type Erasure verwendet wird.

Generische Ausnahmen sind in Java nicht zugelassen. [Ora21b] definiert, dass direkte oder indirekte Subklassen des Interfaces `Throwable` keine generischen Klassen sein dürfen. Diese Einschränkung wird benötigt, da der `catch`-Mechanismus ausschließlich mit nicht-generischen Klassen funktioniert. Somit gilt diese Einschränkung sowohl für die sogenannten *checked*, als auch die für die *unchecked Exceptions*.

Instanziierung von Typparametern kann in Java nicht implementiert werden. Um ein Objekt eines Typparameters zu instanziieren, können mehrere Ansätze gewählt werden. Zwei Möglichkeiten werden bereits in [GH05] dargestellt. So kann ein Array eines Typparameter `T` instanziiert werden, indem zuvor ein Array von `Object` erstellt wird. Abschließend wird dieses in ein Array von `T` umgewandelt. Diese Umwandlung ist jedoch nicht sicher! Somit kann eine `ClassCastException` entstehen. Eine sichere Lösung ist die Verwendung sogenannter *Typ Token*. Diese stellen ein Feld in der generischen Klasse vom Typ `Class<T>` dar. Das entsprechende Feld muss folgend über den Konstruktor gesetzt werden. Mithilfe dieses Feldes, können *factory*-Methoden Instanzen des gewünschten Types erstellen, welche anschließend sicher umgewandelt werden können. Ein Beispiel einer solchen factory-Methode ist die Methode `newInstance(Class<?>`

`componentType, int length)` bzw. deren Überladung. Diese Methode ist Bestandteil der Klasse `Array` aus dem Package `java.lang.reflect`.

Wie zu erkennen ist, bietet die Type Erasure sowohl Vor- als auch Nachteile. Einige Nachteile können dabei einfach, andere gar nicht behoben werden. So stellt die Unfähigkeit der Kompilierung der Klasse `OLFun` aus Listing 2.2 ein Problem für Java-TX dar. Ein Lösungsansatz hierzu wird im Abschnitt 2.5 „Theoretische Lösung zur Übersetzung echter Funktionstypen“ vorgestellt.

2.3. Übersetzung generischer Parameter in Java vergleichbaren Sprachen

Die nachfolgenden Unterabschnitte behandeln die Darstellung der Übersetzungen generischer Parameter in Programmiersprachen, welche mit Java vergleichbar sind. Als vergleichbar werden Programmiersprachen eingestuft, welche auf der JVM ausgeführt werden können (deren Bytecode). Somit werden die Ansätze im Rahmen der Möglichkeiten der JVM betrachtet. Zu betrachten sind nicht nur Programmiersprachen, sondern auch bereits publizierte Erweiterungen von Java. Darunter fällt unter anderem auch Pizza. Um die Darstellung der Möglichkeiten in einen größeren Kontext zu fassen, wird außerdem die Sprache C# mit .NET betrachtet. Hierbei handelt es sich um eine Sprache, welche nicht auf der JVM ausgeführt wird. Aufgrund deren nachgesagten Ähnlichkeiten zu Java, wird diese dennoch betrachtet. Außerdem handelt es sich bei den zu betrachtenden Ansätzen um hauptsächlich objektorientierte Sprachen. Der Grund hierfür liegt in der paradigmatischen Ähnlichkeit der Sprachen. Der Vergleich mit anderen Paradigmen würde eine nähere Betrachtung derer Typsysteme erfordern. Dies kann im Rahmen dieser Arbeit jedoch nicht gewährleistet werden.

2.3.1. Pizza

In [OW97] wird Pizza als Erweiterung für Java vorgestellt. Zu beachten ist hierbei, dass im Jahre der Publikation von [OW97] Java in der Version 1.1 war. Diese Erweiterung führt die folgenden, damaligen neuen Funktionen ein:

- Parametrische Polymorphie
- Funktionen höherer Ordnung
- Algebraische Datentypen

Pizza wird in Java übersetzt. Anschließend kann der Quellcode in Java zu Bytecode kompiliert werden und auf der JVM ausgeführt werden. Dies bedingt eine enge Kopplung von Pizza zu Java. Eine enge Kopplung bewirkt hierbei, dass die Möglichkeiten der Implementierung von Innovationen an den Umfang von Java gebunden sind [OW97]. Positiv ist jedoch, dass somit die vorgestellten Prinzipien in Java übernommen werden können. Des Weiteren bedingt diese Kompatibilität der Sprache, eine Kompatibilität der Bibliotheken. Somit müssen bereits kompilierte Bibliotheken nicht angepasst werden. Dies entspricht somit dem Grundsatz von Java, Abwärtskompatibilität zu wahren.

Im Folgenden wird die damalige neue Funktion, der parametischen Polymorphie, genauer die damit eingeführten parametrischen Typen, näher betrachtet. Außerdem werden die Funktionen höherer Ordnung betrachtet. Deren Umsetzung und Übersetzung sind für die echten Funktionstypen in Java-TX von Relevanz. Zur Übersetzung von generischen Typen bietet Pizza nach [OW97] zwei verschiedene Ansätze: eine *homogene* und eine *heterogene* Übersetzung. Dabei bezeichnet eine homogene Übersetzung diejenige, welche einen einzigen Code, mit einer generellen Repräsentation erzeugt. Dahingegen wird bei der heterogenen Übersetzung für jede Typkombination ein spezifischer Code erzeugt. Laut [OW97] führt dabei heterogener Code zu einer besseren Laufzeit, wohingegen homogener Code kompakter ist. Die Umsetzung dieser Übersetzungen ist nach [OW97] intuitiv und natürlich. Ausgedrückt werden soll damit, dass die Übersetzungen Code erzeugen, welcher auch *manuell* mit den entsprechenden Intentionen entwickelt werden würde. Die Übersetzungen selbst können heterogen verwendet werden. Somit kann ein Programmcode zum Teil homogen und zum anderen Teil heterogen übersetzt werden. Dies ermöglicht eine individuelle Anpassung an die aktuellen Speicherplatz- und Laufzeitanforderungen. [OW97] schlägt zu Beginn vor, die heterogene Übersetzung für primitive Datentypen zu verwenden. Die Referenztypen sollten somit homogen übersetzt werden. Zur Folge hätte dies einen gleichmäßigen Kompromiss aus Performanz und benötigtem Speicherplatz. In Pizza eingeführte Typen weißen laut [OW97] Ähnlichkeiten zu Templates aus C++ auf. Beide erlauben parametrisierte Typen und weißen eine ähnliche Sytax auf. Im Gegensatz zu den Templates aus C++ kann der Typcheck in Pizza direkt bei der Kompilierung erfolgen.

[OW97, S. 2] führt die parametrische Polymorphie anhand eines Beispiels der Klasse `Pair` ein. Diese stellt ein Paar/Tupel dar, welches zwei Attribute des jeweiligen selben Typs besitzt. Neu eingeführt wird die Schreibweise des Typparameters `elem`, welcher den Typ der beiden Attribute repräsentiert. Zu beachten ist hierbei, dass die Schreibweise des Typparameters mit `<T>` gleich wie einige Jahre später in Java ist. Als Funktionalität stellt diese Klasse bereit, die beiden Attribute zu vertauschen. Es handelt sich somit um ein akademisches Beispiel. Zum Test wurde anschließend die Klasse `Pair` mit den Typargumenten `int` und `String` versehen. So wurden zwei unterschiedliche Instanzen des jeweiligen

parametrisierten Typs `Pair` erstellt. Da sich die jeweiligen Instanzen ausschließlich im Typparameter, aber nicht in der Klasse selbst unterscheiden, beschreibt [OW97, S. 2] dies als *parametrische Polymorphie*. Diese Polymorphie kann laut [OW97, S. 2] auf zwei unterschiedliche Arten von Java simuliert werden. Als erster Ansatz wird die Generierung einer neuen Variante der Klasse `Pair`, für jede instanziierte Parameterkombination vorgestellt. Da jede Kombination individuellen Code erzeugt, wird dieser Ansatz als *heterogene* Übersetzung bezeichnet. [OW97, S. 3] stellt dabei den Ansatz vor, ein erweiterte Klasse mit dem Namen *`Basisklasse_Typargument`* zu generieren. Dabei muss außerdem jede Typannotation des Typparameters mit dem Typargument ersetzt werden. Im Beispiel der Klasse `Pair` führt dies zu den erweiterten Klassen `Pair_int` und `Pair_String`. Die Implementierungen beinhalten dabei statt des Typparameters `elem` die Typen `int` und `String`. Der zweite Ansatz zur Übersetzung wird als *homogene* Übersetzung bezeichnet. Dabei wird nach [OW97, S. 3] der Typparameter durch die Klasse `Object` ersetzt, welche die Basisklasse aller Klassen in Java repräsentiert. Durch die Klassenhierarchie in Java, kann jeder Referenztyp in den Typ `Object` umgewandelt und anschließend wiedergewonnen werden. Nicht-Referenztypen existieren in Java, werden jedoch durch deren korrespondierenden Referenztyp eingeschlossen. So wird der Typ `int` in den korrespondierenden Typ `Integer` überführt.

Wie auch in Java (Abschnitt 2.1 „Generische Parameter in Java“), können in Pizza Typparameter als gebundene Typparameter verwendet werden. Somit bietet Pizza laut [OW97, S. 3] die Möglichkeit der *gebundenen parametrischen Polymorphie*. Dabei stellt der Typparameter einen Typ dar, welcher eine Spezialisierung einer Klasse oder die Implementierung eines Interfaces ist. Außerdem können in Pizza Interfaces ebenso parametrisiert werden. Hierzu werden die Schlüsselwörter `implements` und `extends` bei der Parameterdeklaration verwendet [OW97]. Als erweitertes Beispiel dient der Quellcode aus Listing L.2 von [OW97, S. 3]. Die Klasse `Pair` erhält einen gebundenen Typparameter `elem`. Somit repräsentiert `elem` einen Typ, welcher das Interface `Ord` implementiert. Hierbei tritt laut [OW97, S. 3] eine sogenannter *F-gebundenen* Polymorphie auf. Diese besagt, dass der Typparameter auch in der Grenze (hier `Ord<`*`elem`*`>`) verwendet wird und somit eine rekursive Grenze deklariert wird. Die Funktionalität dieses Codes besteht darin, durch die Methode `min` der Klasse `Pair` das kleinere Attribut zu erhalten. Für die Implementierung wird dabei die Methode `less` des Interfaces `Ord` verwendet. Auch existieren nach [OW97, S. 3 f.] die beiden Ansätze zur Übersetzung der Parameter. So kann eine heterogene Übersetzung analog zu den ungebundenen Parametern erfolgen. Dabei werden erneut neue Varianten der parametrisierten Klassen generiert und der Typparameter in diesen durch den konkreten Typ ersetzt. Da Pizza ausschließlich Kovarianz in den Typparametern gestattet, muss keine weitere Anpassung der Typen im Quelltext erfolgen. Im Gegensatz zur heterogenen Übersetzung, muss die homogene angepasst werden. Ungebundene Typparameter werden

erneut durch den Basistyp `Object` ersetzt. Neu ist anschließend, dass die gebundenen Typparameter mit deren Grenze und nicht mit dem Basistyp `Object` substituiert werden. Die konsequente Durchführung dessen führt jedoch zu inkompatiblen Methodensignaturen. So wird der Typ der Methode `less` des Interfaces `Ord` zu:

$$\texttt{Object} \rightarrow \texttt{boolean}.$$

Im Gegensatz dazu weißt die Signatur `less` der Klasse `OrdInt` die Signatur

$$\texttt{OrdInt} \rightarrow \texttt{boolean}$$

auf. Der entsprechende Quellcode ist in Listing L.3 dargestellt. Um diesen Umstand zu beheben, wird die Methode `less` des Interfaces `Ord` zu `less_Ord` umbenannt. Die Klasse `OrdInt`, welche das Interface `Ord` implementiert, behält dessen originale Methode `less`. Hinzugefügt wird jedoch die Methode `less_Ord`, welche als *Brückenmethode* fungiert, welche die Methode `less` aufruft, nach der Umwandlung des Parameters. [OW97, S. 4] erwähnt dabei, dass das nicht benötigte Interface `Ord_OrdInt` entfernt werden kann. Diese bietet als spezielles Interface nicht die Generalisierung, für welche ein Interface ursprünglich vorgesehen ist. Wie zu erkennen ist, wurde in Java bei der Einführung der generischen Parameter, eine Art der hier vorgestellte homogene Übersetzung implementiert. Auch in Java wird die Grenze eines gebundenen Typparameters an dessen Stelle eingesetzt. Dies kann durch die Deskriptoren entsprechend verifiziert werden. Zudem erstellt auch Java Brückenmethode, um das Problem der inkompatiblen Methoden zu lösen. Auch dies kann anhand der Dekompilation von Java Bytecode verifiziert werden.

[OW97, S. 5 f.] führt außerdem Funktionen höherer Ordnung in Pizza ein. Dabei existiert erneut eine heterogene Übersetzung dieser. Die heterogene Übersetzung generiert eine abstrakte Klasse, welche die Signatur der Funktion deklariert. Somit stellt diese den Funktionstyp dar. Für jede funktionalen Abstraktion wird wiederum eine eigene Klasse erstellt. Im Gegensatz dazu wird bei der homogenen Übersetzung erneut eine Substitution mit dem Basistyp `Object` durchgeführt. Hierbei werden alle formalen Parameter und der Rückgabetyp mit dem Typ `Object` annotiert. Somit stellt die homogene Übersetzung nach [Ora19, S. 7] kompakteren Code, im Gegensatz zur heterogenen Übersetzung, bereit. Dafür verliert die homogene Übersetzung Informationen der korrekten statischen Typen. Deshalb müssen während der Laufzeit mehr Typüberprüfungen durchgeführt werden.

2.3.2. Scala

Entwicklung und Einflüsse

[OSV08] stellt Scala als Programmiersprache vor. Scala steht für „scalable language“, da Scala mit den steigenden Anforderungen wachsen soll. So kann diese Programmiersprache zum Verfassen von einfachen Skripten, bis hin zu komplexen Softwareprojekten verwendet werden. Dabei kann Scala auf der JVM ausgeführt werden und dabei Java Bibliotheken verwenden. Somit ist Scala interoperabel mit Java. Laut [OSV08] vereint Scala die Konzepte der Objektorientierung und der funktionalen Programmierung. Diese Programmiersprache wurde explizit für diese Vereinigung konzipiert. Sie vereint nicht diese beiden Paradigmen im Nachgang, wie Java (siehe Abschnitt 2.4 „Neue Funktionalitäten in Java-TX “). Dennoch weißt Scala genau wie Java die Eigenschaft der statischen Typisierung auf. Durch die Kombination der beiden Paradigmen können nach [OSV08] die jeweiligen komplementären Stärken additiv verwendet werden. Als weitere Gründe für den Einsatz von Scala nennt [OSV08] die Kompatibilität zu Java, da Scala Programme in Bytecode kompilieren. Dieser Aspekt ist von großer Relevanz für die Betrachtung der generischen Parameter und deren Umsetzung im Bytecode. Außerdem sind Scala Programme kurz, da dieses ca. nur halb so viele Codezeile, wie ein funktional äquivalentes Java Programm aufweist. Zudem wird in dieser Programmiersprache auf einem hohen Level programmiert, welches sich wiederum in der Kürze des Codes widerspiegelt. Als letzter Grund wird in [OSV08] das Typsystem angebracht. Wie auch Java, bietet Scala generische Typen an. Außerdem umfasst Scala Intersektionstyp und abstrakte Typen.

Laut [OSV08] wurde die Entwicklung von Scala historisch von bereits bestehenden Programmiersprachen beeinflusst. Außerdem umfasst diese einige neue Konstrukte, welche jedoch wiederum nachträglich Einzug in weitere Programmiersprachen hielten. Dabei führten hauptsächlich die Zusammensetzung der Komponenten zu den Innovationen in der Sprache selbst. Nachfolgend werden einige dieser Einflüsse in Auszügen aus [OSV08] erwähnt. So bedient sich Scala der Syntax von Java und C# . Außerdem werden Blöcke, Aussagen und Ausdrücke in Scala fast wie in Java verfasst. Des Weiteren wird die Syntax von Klassen, Paketen und `import`-Anweisungen von Java übernommen. Zudem bedient sich Scala der Ausführungseinheit (JVM), der Bibliotheken und der Basistypen von Java. Klar zu erkennen ist somit, dass Java einen sehr großen Einfluss auf die Entwicklung von Scala hatte. Ansätze wie das universelle einbetten von beinahe allen Konstrukten wurde bereits von Algol der Simula eingeführt. Scala verband zudem nicht als erste Sprache die Ansätze zweier Paradigmen, wobei laut [OSV08] Scala am weitesten geht. So setzen auf der Java-Platform diese Ansätze bereits die Sprachen Pizza, Nice oder auch Mulit-Java um. Wie zu erkennen ist, wirkte somit Odersky bei der Entwicklung von Pizza und nun

auch bei der Entwicklung von Scala mit. Zudem existieren funktionale Sprachen, welche Eigenschaften der Objektorientierung hinzufügten, wie OCaml. Neuerungen die in Scala umgesetzt wurden waren beispielsweise die abstrakten Typen, welche eine bessere objektorientierte Alternative zu generischen Typen liefern.

Generische Parameter

[Ode+22] umfasst die Sprachdefinition von Scala. Für die generischen Typen bzw. generische Parameter ist hiervon das Kapitel 3 „Types“ bzw. genauer die Unterpunkte 2.6. und 7. von Nöten. Die Informationen dieses Absatzes wurden aus dieser Quelle bezogen. Ein parametrisierter Typ wird in der Form T`[`T_1`, ... ,`T_n`]` mit $n \geq 1$ definiert. Dabei stellt T den Bezeichner eines Typenkonstrukts, welches n Typparameter umfasst. Als wohl geformt gilt ein parametrisierter Typ, sollten alle tatsächlichen Typparameter (die Typargumente) eine Spezialisierung der jeweiligen oberen Grenze und eine Generalisierung der jeweiligen unteren sein. Somit werden parametrisierte Typen ähnlich wie in Java definiert. Der Unterschied besteht dabei in den speziellen Zeichen, welche die Liste der Typparameter umschließt. Zudem weißen Scala und Java eine Ähnlichkeit der generischen Parameter auf, da auch in Scala hierfür eine *Type Erasure* existiert. In Scala wird ein Typ als generisch definiert, sollte dieser Typargumente oder Typvariablen aufweisen. Die Type Erasure definiert (wie auch in Java), eine Abbildung von potentiell generischen Typen auf nicht generische Typen. Eine Definition des *Erasure-Operator* erfolgt zudem analog wie in Java. Die Abbildungen wurden ebenso analog zu denen in Java definiert. Unterschiede liegen jedoch in den zusätzlichen Abbildungen für Konstrukte, welche nicht in Java vorhanden sind.

Somit kann zusammen gefasst werden, dass Scala zwar auf der JVM mittels Bytecode ausführbar ist, jedoch selbst der Type Erasure aufgrund dessen unterliegt. Dies führt auch in Scala dazu, dass Methoden welche sich ausschließlich in den Namen der Typvariablen unterscheiden, nicht in deren Signaturen unterscheiden. Dennoch kann Scala solch ein Programm kompilieren und ausführen. Ein Beispiel hierfür ist die Klasse `OLFun` aus Listing 2.3. Das Interface `Fun1$$`, welches in Java-TX für Funktionen verwendet wird, wurde als `trait` analog definiert. Somit handelt es sich bei der Klasse `OLFun` um eine funktional äquivalente Implementierung zu dem Code aus Listing 2.2 in Java-TX .

```
class OLFun {
  def m(f:Fun1$$[Double,Double]):Double = f
      .apply(1)
  def m(f:Fun1$$[Int,Int]):Int = f.apply(1)
  def m(f:Fun1$$[String,String]):String = f
      .apply("Hello World")
```

```
  //def m(f:Fun1$$[Int,Int]):Float = 2.0f
  //def m(f:Fun1$$[Float,Float]):Double = 1
}
```

Listing 2.3: Klasse `OLFun` in Scala mit generischem Typ `Fun1$$` als formaler Parameter.

Aufschluss darüber, weshalb dieser Code kompiliert und ausgeführt werden kann, liefert die auskommentierten letzten Deklarationen der Methode `m`. Sollte diese Kommentare entfernt und die Deklarationen verwendet werden, so werden die folgende Fehler beim Kompilieren ausgelöst:

```
1)
method m is defined twice;
the conflicting method m was defined at
    line 3:7

2)
double definition:
def m(f: [Double,Double]): Double at line 2
     and
def m(f: [Float,Float]): Double at line 7
have same type after erasure: (f: ): Double
```

Somit ist zu erkennen, dass Scala den Code derart kompiliert, dass die JVM die Methoden, basierend auf dem gesamten Deskriptor, auflösen kann. Um genauer zu spezifizieren, kann Scala solche Überladungen auflösen, welche sich sowohl in *mindestens einem Typargument und dem Rückgabetyp* unterscheiden. Diese Hypothese besteht, da der Bytecode einer äquivalenten Klasse in Java, welche ausschließlich eine dieser Methodendeklarationen besitzt, keine Unterschiede beim Methodenaufruf bzw. der Signatur verzeichnet. Möglich ist diese Kompilierung jedoch nur bei generischen Typen. So kann beispielsweise der Typ `Fun1$$` durch den Typ `List` ersetzt werden und dennoch kompiliert Scala diesen Code. Sollte jedoch diese Überladung mit einem beliebigen Referenztypen oder primitiven Typen als formaler Parameter deklariert werden, so schlägt die Kompilierung fehl. Der Java Compiler `javac` lässt eine solche Überladung, welche ausschließlich im Rückgabetypen stattfindet, nicht zu. Daher löst Scala zwar das Problem des Codes aus Listing 2.2 mit den Typen aus Abbildung 2.3, aber bietet keine generelle Lösung für die Type Erasure. Wie in Listing 2.3 dargestellt, könnten dennoch weitere sinnvolle Überladungen nicht kompiliert werden, wie die letzte auskommentierte Deklaration.

2.3.3. C#

Einführung C# und .NET

In [Mic21a] wird eine Einführung in die Programmiersprache C# gegeben. Bei C# handelt es sich um eine objektorientierte und typsichere Programmiersprache, welche von Microsoft entwickelt wurde. Ausgeführt wird diese dabei auf .NET . Wie der Name vermuten lässt, liegen die Wurzeln dieser Sprache in den C ähnlichen Sprachen. Im Kern wurde C# als objektorientierte Sprache entwickelt, jedoch erweiterte sich diese mit der Zeit um weitere Konzepte. So weißt diese heute Funktionalitäten aus weiteren Paradigmen auf. C# ist nicht nur objektorientierte, sondern auch komponentenorientiert und somit nach [Mic21a] dazu geeignet, um Softwarekomponenten zu entwickeln. Einige nützliche Funktionalitäten wie die *Garbage collection*, Null-Typen oder auch Ausnahmebehandlungen weißt C# auf. Aus der funktionalen Programmierung wurden Lambda-Ausdrücke übernommen. Außerdem bietet C# laut [Mic21a] mit Language Integrated Query (LINQ) Möglichkeiten zur einheitlichen Bearbeitung von Daten an. Die Syntax hiervon orientiert sich dabei stark an der von relationalen Sprachen wie SQL. Somit kann angenommen werden, dass C# relationale Funktionalitäten besitzt. Zudem besitzt C# nach [Mic21a] ein *einheitliches Typsystem*. Dies bedeutet, dass alle Typen einen gemeinsamen Basistyp, den Typ `object` besitzen. Darin eingeschlossen sind auch primitive Typen. Dadurch wird eine konsistente und einheitliche Verarbeitung und Speicherung der Daten gewährleistet. Nach [Mic21a] können Programmierende sowohl Referenz-, als auch Werttypen definieren. Zudem können generische Methoden und Typen definiert werden.

C# wird laut [Mic21a] auf .NET ausgeführt. Bei .NET handelt es sich ähnlich wie die JVM um eine virtuelle Ausführungssystem, die sogenannte Common Language Runtime (CLR). Die CLR stellt dabei Microsofts Implementierung der Common Language Infrastructure (CLI) dar, welche wiederum ein internationaler Standard ist. CLI stellt Vorgaben für Ausführungs- und Entwicklungssysteme, sodass Sprachen und Bibliotheken ohne Komplikationen zusammenarbeiten können. Nach [Mic21a] wird der Quellcode von C# in ein Zwischencode kompiliert, welcher den Vorgaben der CLI entspricht. Dieser Code und die benötigten Ressourcen werden dabei in einer sogenannten *Assembly* abgespeichert. Bei der Ausführung wird eine solche Assembly in die CLR geladen und mittels eines JIT in nativen Code übersetzt. Die CLR wiederum bietet weitere Funktionalitäten während der Ausführung, wie Garbage collection oder eine Ressourcenverwaltung an. Code welcher auf der CLR ausgeführt wird, wird als *verwalteter Code* bezeichnet. Eine Haupteigenschaft von .NET ist laut [Mic21a] die Interoperabilität zu weiteren Sprachen. So kann Zwischencode von C# mit Zwischencode aus Visual Basic oder auch C++ interoperieren. Daher kann eine Assembly aus Modulen verschiedener Programmiersprachen bestehen, welche wiederum miteinander interagieren, als seien sie in der selben entwickelt worden.

Generische Parameter

Da der Programmcode von C# in Zwischencode kompiliert wird, welcher wiederum auf der CLR .NET ausgeführt wird, werden die generischen Parameter in .NET direkt betrachtet. Als Beispiele für z.B. die Notation wird jedoch die Sprache C# verwendet. Laut [Mic21b] können generische Typen in .NET sowohl Klassen, als auch Strukturen und Interfaces sein. Zudem besteht die Möglichkeit der Definition von generischen Methoden. Typvariablen werden wie auch in Java stellvertretend für die Typen von Feldern oder in der Methodensignatur verwendet. Deklariert werden die Typvariablen als Typparameter in der Klassendeklaration. Die Syntax in C# ähnelt der von Java. Bei der Generierung einer Instanz einer generischen Klasse, müssen die Typargumente angegeben werden. Diese ersetzen dabei die Typvariablen durch die korrekten Typen. Somit entsteht nach [Mic21b] eine neue generische Klasse, welche auch als *konstruierte generische Klasse* bezeichnet wird. Konstruierte generische Klassen besitzen die Eigenschaft der Typsicherheit. Es sind somit die Typen der Typargumente zur Kompilierzeit und Laufzeit bekannt. Daher müssen weniger Typumwandlungen durchgeführt werden und die Wahrscheinlichkeit von Laufzeit-Typfehlern wird reduziert. Der Aufwand der Gewährleistung der Typsicherheit wird somit vom Compiler getragen. Somit liegt keine *Type Erasure* in C# , genauer bei .NET vor.

Der erste Entwurf und die erste Implementierung von generischen Typen in .NET wurden dabei bereits 2001 in [KS01] publiziert. [KS01, S. 1] beschreibt zuerst die Gründe dafür, dass das virtuelle Ausführungssystem generische Parameter unterstützen muss. Denn sollte das virtuelle Ausführungssystem generische Parameter nicht unterstützen, so führt dieses zu erheblichen Einschränkungen. Als Beispiele werden Leistungseinbußen in Pizza oder komplizierte Kompilierungsverfahren in NextGen genannt. Die Polymorphie aus [KS01, S. 1] soll sowohl die Instanziierung von Referenz-, als auch Werttypen und zugleich exakte Laufzeittypen unterstützen. Um diese Anforderungen umzusetzen, reicht nach [KS01, S. 1] nicht nur die Kompilierung aus und es muss eine Unterstützung des virtuellen Ausführungssystems existieren. Die Implementierung setzt diese Funktionalitäten im wesentlichen durch zwei Ansätze um [KS01, S. 2]. Instanzen parametrisierter Klassen werden dynamisch geladen und der Code für deren Methoden wird erst vor der Verwendung generiert. Somit handelt es sich um eine *just-in-time Spezialisierung* der generischen Klasse. Des Weiteren werden, falls möglich, der Code oder Datendarstellungen zwischen verschiedenen Instanzen geteilt. Durch die Typspezialiersung muss außerdem keine Umwandung von primitiven Datentypen, in der Implementierung, vorgenommen werden. Dies führt nach [KS01, S. 3] dazu, dass diese Implementierung effizienter ist, als wenn die primitiven Datentypen in den Typ `object` umgewandelt und darin gespeichert werden müssten. Eine genaue Beschreibung der Implementierung kann in [KS01] nachgelesen werden. Im Rahmen dieser Arbeit wird dies Implementierung jedoch nicht näher erläutert.

2.4. Neue Funktionalitäten in Java-TX

In den vergangenen Versionen von Java konnte beobachtet werden, dass Konzepte unterschiedlicher Paradigmen darin Einzug hielten. So wurden in Java 8 Lambda Ausdrücke eingeführt [Ora14]. Durch diese könne Funktionen als Argumente einer Methode verwendet werden. Des Weiteren kann dadurch eine Funktionalität als Daten behandelt werden, indem diese einem Attribut bzw. einer Variablen zugewiesen wird. Nach [Ora21b] beinhaltet ein Lambda Ausdruck, wie „normale Methoden“, formale Parameter und einen Funktionsrumpf. Dabei kann der Funktionsrumpf entweder aus einem Ausdruck oder aus einem Block bestehen. Notiert werden Lambda Ausdrücke dabei nach dem Schema *`Parameterliste -> Lambda Rumpf`*. Eine Einschränkung dieser besteht jedoch im Kontext derer Verwendung. Lambda Ausdrücke können nur im Kontext einer Zuweisung, eines Aufrufs oder einer Umwandlung verwendet werden [Ora21b]. Somit muss diesen durch den entsprechenden Kontext ein passender Typ zuweisbar sein. In einem sogenannten *String Kontext* oder *Numeric Kontext*, können Lambda Ausdrücke nicht verwendet werden. Dies würde zu einem Kompilierfehler führen. Die Auswertung eines Lambda Ausdrucks führt nicht augenblicklich zur Ausführung des Rumpfes. Eine Ausführung wird laut [Ora21b] erst durchgeführt, wenn eine entsprechende Funktion aufgerufen wird. Somit setzten Lambda Ausdrücke die sogenannte *Lazy Evaluation* um. Dennoch führt diese Auswertung zur Generierung einer Instanz eines funktionalen Interfaces [Ora21b]. Das funktionale Interface ist in Java definiert als Interface, welches ausschließlich eine abstrakte Methode besitzt und nicht als `sealed` deklariert wurde. Typisiert sind Lambda Ausdrücke dabei durch sogenannte Zieltypen, durch die jeweiligen Kontexte.

Eine weitere Funktionalität, welche in Java eingeführt wurde, ist das Pattern Matching. Dieses wird in [Ora21b] als Test beschrieben, welcher auf einem Wert ausgeführt wird. Dabei taucht das Pattern als Operand eines Ausdrucks oder einer Aussage auf, welche den zu testenden Wert umfassen. Ein Pattern kann zudem eine lokale Variable einführen, welche als *Pattern-Variable* bezeichnet wird. Nach dem Stand von [Ora21b] existiert aktuell nur eine Art von Pattern, und zwar *Typ-Pattern*. Diese werden genutzt um zu überprüfen, ob der Wert eine Instanz der im Pattern enthaltene Typ ist. Die Mechanik des Pattern Matchings beschränkt sich somit darauf, eine Typprüfung durchzuführen. Zu wahr (`true`) evaluiert das Pattern Matching, sollte es sich bei dem Wert nicht um `null` handeln und der Wert ohne `ClassCastException` zum Typ des Typ-Pattern umgewandelt werden kann. Ansonsten evaluiert dieses zu unwahr (`false`). Einsatz findet das Pattern Matching nach [Ora21b] im `instanceof` Operator oder in `switch`-Ausdrücken. Somit können diese nicht in der Methoden- bzw. Funktionssignatur wie in Haskell [Jon02] verwendet werden.

Wie zu erkennen ist, entwickelt sich die Programmiersprache Java immer weiter. Dabei werden sowohl inter-, als auch intraparadigmatische Funktionalitäten hinzugefügt. Beispiele

für intraparadigmatische Funktionalitäten wären beispielhaft die Lambda Ausdrücke oder das Pattern Matching. Java-TX versucht dabei, weitere Funktionalitäten hinzuzufügen. Diese werden im Folgenden näher erläutert.

Echte Funktionstypen

Die Einführung von echten Funktionstypen in Java-TX wurde in [PS17] betrachtet. Hierbei wurden zuerst die bestehenden Vor- und Nachteile der aktuellen Implementierung von Lambda Ausdrücken seit Java 8 beschrieben. Als größter Vorteil für die kompatiblen Zieltypen wurde dabei nach [Goe13] die einfache Möglichkeit zur Realisierung von *Rückruffunktionen (eng. callback functions)* angegeben. In einigen Application Programming Interfaces (APIs) werden Rückruffunktionen durch *Rückrufinterfaces (eng. callback interfaces)* realisiert. Die Programmierenden müssen anschließend die Rückruffunktion als Instanz des Rückrufinterfaces einer Methode übergeben. Dies geschieht dabei meist durch anonyme Klassen. Ein bekanntes Beispiel hierfür wäre das Rückrufinterface `ActionListener`. Nach [PS17, S. 3] führt die Implementierung von Lambda Ausdrücken mit Zieltypen dazu, dass entsprechende APIs mit Lambda Ausdrücken aufrufbar sind. Dabei muss kein bereits existierender Bytecode in den Bibliotheken angepasst werden, da sich der Zieltyp der Lambda Ausdrücke diesen „anpasst“. Somit wird impliziert, dass ein konkreter Typ für die Lambda Ausdrücke dazu geführt hätte, dass die Bibliotheken hätten angepasst werden müssen. Ansonsten wären diese nicht zu Lambda Ausdrücken kompatibel. Im Gegensatz dazu zeigt [PS17, S. 4] zwei Nachteile dieses Ansatzes auf. Der erste davon ist die direkte Anwendung von Lambda Ausdrücken. Nach dem λ-Calculus muss die Definition eines Lambda Ausdrucks mit anschließender Evaluation mit gegebenen Argumenten möglich sein. In Java 8 müsste somit der Code `(x -> h(x)).apply(arg);` korrekt sein. Dies ist jedoch in Java aufgrund der Zieltypen nicht möglich, da es sich um keinen korrekten Kontext für den Lambda Ausdruck handelt. Daher kann kein korrekter Zieltyp inferiert werden und die Methode `apply` wird nicht korrekt identifiziert [PS17]. Um dies zu umgehen, kann durch die Umwandlung des Lambdas ein Umwandlungskontext geschaffen werden. Der Code müsste nach [PS17, S. 4] zu `(Function<T,R> x -> h(x)).apply(arg);` angepasst werden. Dies führt jedoch bei Funktionen in der curry-Schreibweise zu einer unübersichtlichen Notation. Ein weiterer Nachteil besteht in der Spezialisierung von Funktionen durch Lambda Ausdrücke. [PS17, S. 4] fasst die aktuelle Umsetzung dessen wie folgt zusammen: Die Spezialisierung kann durch die Interfaces im Packet `java.util.function` simuliert werden. Dazu muss die Kontravarianz der Argumenttypen und die Kovarianz der Rückgabetypen mithilfe der Super-Wildcard und Extends-Wildcard umgesetzt werden. Außerdem müssen Typumwandlungen annotiert werden, wie auch bei der direkten Anwendung von Lambda Ausdrücken. Zusammen führt dies zwar zur erfolgreichen Simulation der Spezialisierung,

jedoch gleichzeitig zu unleserlichem und verworrenem Code. Aus diesen Gründen wurde in [PS17] beschrieben, wie die Vorteile von Zieltypen beibehalten, jedoch die Nachteile beseitigt werden können.

Zur Umsetzung dessen wurden in Java-TX nach [PS17, S. 8] echte Funktionstypen basierend auf dem Ansatz aus Scala eingeführt. Dazu wurden zwei neue Interfaces `Fun`N`*<-`A_1`, ..., -`A_n`, R>` und `FunVoid`N`*<-`A_1`, ..., -`A_n`>` eingeführt. Das N gibt die Anzahl Typparameter an, welche für die Argumente der Methode `apply` in den Interfaces genutzt wird. `Fun`N besitzt zudem einen weiteren Typparameter für den Rückgabetyp von `apply`, wohingegen dieser bei `FunVoid`N `void` ist. Somit besitzt diese keinen Rückgabetyp. Das `-` vor den Typparametern steht für die Kontravarianz der Argumenttypen und das `+` für die Kovarianz des Rückgabetyps. Diese Varianzen können in Java jedoch nicht ohne weiteres übernommen werden, da Java ausschließlich die sogenannte *use-site* Varianz und nicht die benötigte *declaration-site* Varianz unterstützt. Aus diesem Grund wurde diese Varianz in der Implementierung vernachlässigt. Zum Erhalt dieser Varianzen ist keine use-site Varianz für diese Interfaces erlaubt. Durch die explizite Typisierung von Lambda Ausdrücken durch diese Interfaces, wurden die beiden Nachteile laut [PS17, S. 8] behoben. Außerdem definiert [PS17, S. 8 f.] die Kompatibilität dieser Interfaces mit Zieltypen. Die Konsequenz aus dieser Kompatibilität besteht darin, dass als Rückruffunktionen Lambda Ausdrücke, welche mit den Funktionstypen typisiert wurden, verwendet werden können. So kann eine Funktion mit dem Typ `Fun1*<ActionEvent, String>` für eine Rückruffunktion des Zieltyps `ActionListener` verwendet werden. In der aktuellen Version von Java-TX wurden die beiden unterschiedlichen Interfaces als Funktionstypen durch ein „einheitliches“ ersetzt. Funktionen besitzen somit den Typ `Fun`N`$$`. Dabei steht N für die Anzahl der Typparameter für die Argumente der Methode `apply`. Zusätzlich enthält `Fun`N`$$` einen weiteren Typparameter für den Rückgabetyp der Methode `apply`. Sollte dieser `void` sein, so muss als Typargument die Klasse `Void` verwendet werden. Zu beachten ist dabei, dass die Methode `apply` dennoch eine `null`-Referenz zurück gibt, da ausschließlich diese mit dem Typ `Void` korrespondiert. Aufgrund dieser Änderung wurde in dieser Arbeit ausschließlich der Typ `Fun`N`$$` als Funktionstyp verwendet.

Globale Typinferenz

Die traditionelle Java Philosophy beinhaltet laut [UW18], das alles, selbst simple Ausdrücke, mit statischen Typen deklariert werden müssen. Dies kann dazu führen, dass komplexe und unleserliche Typen deklariert werden müssen. Außerdem führt die steigende Anzahl von APIs, welche generische Parameter nutzen, dazu dass die Typen noch ausführlicher und unleserlicher werden. Auf der anderen Seite bilden diese Typen einen starken Vertrag, welcher in der Instandhaltung und Weiterentwicklung eingehalten werden muss. Dennoch

wurde bereist in der Vergangenheit von Java dieses Problem adressiert und versucht zu vereinfachen. Hierzu wurden nach [UW18] verschiedene Anwendungsbereiche für die Typinferenz gefunden, um kürzeren Quellcode zu ermöglichen. Die Typinferenz beschreibt dabei informell den Umstand, dass der Compiler die statischen Typen berechnen und einfügen kann.

In Java 5 wurden nach [UW18] mit den generischen Parametern eingeführt, dass die Typargumente einer generischen Methode inferiert werden können. So kann laut diesem anstelle des Quellcodes `List<String> cs = Collections.<String>emptyList();` der Code `List<String> cs = Collections.emptyList();` verwendet werden. Hierbei wird das Typargument `String` für die Methode `emptyList` vom Compiler bestimmt. [UW18] beschreibt die Typinferenz von Ausdrücken seit Java 7. So können die Typragumente eines Ausdrucks basierend auf dessen Kontext bestimmt werden. Der Quellcode kann somit von `Map<String, List<String>» myMap = new HashMap<String,List<String>»();` zu `Map<String, List<String>» myMap = new HashMap<>();` vereinfacht werden. Dabei kann der Compiler die Typargumente des `HashSet` durch die Typdeklaration auf der linken Seite der Zuweisung bestimmen. Der sogenannte *Diamant Operator* `<>` wurde für diesen Zweck eingeführt. Die Idee, die Typen basierend auf deren Kontext zu inferieren, wurde außerdem in den Lambda Ausdrücken in Java 8 umgesetzt. Daher können Lambda Ausdrücke entweder als `Function<Integer,Integer> f = (Integer x) -> x++;` mit Typdeklaration oder als `Function<Integer,Integer> f = x -> x++;` ohne Typdeklaration formuliert werden. Zusammengefasst führen diese Funktionalitäten zu einer besseren Lesbarkeit und Verständlichkeit des Quellcodes. Zudem wird die Flexibilität des Codes ein wenig erhöht, da durch Änderung des Typs im Kontext, keine Anpassungen der Ausdrücke bzw. Typdeklarationen vorgenommen werden müssen.

Um diesen Gedanken der Erhöhung der Flexibilität weiter zu fördern, wurde mit Java 10 die Typinferenz für lokale Variablen eingeführt [UW18]. Außerdem kann hierdurch die Lesbarkeit des Quellcodes erhöht werden. Sprachen wie Scala oder C# erlauben deshalb die Verwendung des `var` Schlüsselwortes anstelle eines Typen in der Deklaration einer lokalen Variable. Dabei inferiert der Compiler den passenden Typ anhand der Variableninitialisierung. Ein Beispiel hierfür wäre nach [UW18]: `var userChannels = new HashMap<User, List<String>»();`. Außerdem kann `var` verwendet werden, wenn eine lokale Variable mit dem Rückgabewert einer Methode initialisiert wird. Das `var` Schlüsselwort kann zudem dazu verwendet werden, um nicht-bezeichenbare Typen zu inferieren [UW18]. Ein Beispiel hierfür wäre die Inferenz einer anonymen Klasse, welche die Klasse `Object` um ein Attribut erweitert. Ohne `var` müsste für diese der Typ `Object` verwendet werden, wobei das zusätzliche Attribut nicht identifizierbar wäre und somit nicht verwendbar. Durch die Inferenz der anonymen Klasse in `var`, kann dieses Attribut jedoch genutzt werden. Zu beachten ist dabei, dass laut [UW18] die Variablen dennoch einen

statischen Typ besitzen. So führt eine Zuweisung eines Wertes, mit inkompatiblem Typ, zu einem Kompilierfehler. Zudem wird der spezifische Typ eines Ausdrucks als inferierter Typ verwendet. Dies bedeutet, dass die lokale Variable `var b = new B();` mit `B <: A` und `C <: A` den Typ `B` besitzt. Eine Folge hiervon ist, dass eine Zuweisung wie `b = new C();` in einem Kompilierfehler endet. [UW18] fasst dieses damit zusammen, dass polymorpher Code mit dem `var` Schlüsselworte nicht gut harmonisiert. An einer solchen Stelle muss somit dennoch eine explizite Typdeklaration erfolgen. Dies ist jedoch nicht die einzige Einschränkung der lokalen Typinferenz. Wie der Name aussagt, handelt es sich um eine Inferenz im lokalen Kontext. Somit kann diese nach [UW18] nicht zur Inferenz von Typen von Attributen oder Methodensignaturen verwendet werden. Außerdem benötigt die lokale Typinferenz eine direkte Variableninitialisierung bei der Variablendeklaration. Dies hat zur Folge, dass der Code `var x;` zu einem Kompilierfehler führt. Ähnlich hierzu ist die Initialisierung einer Variablen mit der `null` Referenz. Da `null` bei der Initialisierung mit jedem möglichen Typ assoziiert werden kann. [UW18] beschreibt außerdem, dass `var` nicht für Lambda Ausdrücke verwendet werden kann. Lambda Ausdrücke benötigen in Java einen Kontext, durch welchen deren Zieltyp bestimmt werden kann. Das Schlüsselwort `var` stellt dabei keinen solchen expliziten Zieltyp dar. Im Gegensatz dazu kann ab Java 11 das `var` Schlüsselwort selbst in den formalen Parametern eines Lambda Ausdruckes verwendet werden. So kann der Code `(@Nonnull var x, var y) -> x.process(y)` verwendet werden. Dies führt dazu, dass der formale Parameter zwar inferiert wird, dieser jedoch auch annotiert werden kann [UW18].

Bevor Java 10 die lokale Typinferenz einführte, wurde in [Plü07] eine Typinferenz vorgestellt, welche sowohl lokale Variablen, als auch Methodensignaturen inferiert. Nach aktuellem Stand von Java-TX wurde diese Typinferenz zu einer globalen Typinferenz ausgeweitet. Somit können jegliche Typannotationen in Java-TX vernachlässigt werden. Bei der Typinferenz in [Plü07] wird der Typinferenz Ansatz von Hindley-Milner verwendet, welcher vorsieht prinzipale Typen zu inferieren. Dies steht somit im Gegensatz zum Ansatz von Palsberg und Schwartzbach, welcher die spezifischsten Typen inferiert [Plü07, S. 7]. Der Typinferenzalgorithmus beginnt damit, für jeden Ausdruck, Aussage, Block und Methode Typvariablen als TPHs zu vergeben. Dabei wird angenommen, dass eine Methode als Funktion behandelt werden kann, welche wiederum TPHs für die formalen Parameter und den Rückgabetyp erhält. Anschließend wird der abstrakte Syntaxbaum durchlaufen und infolgedessen werden die Typen der TPHs bestimmt. Hierbei werden in jedem Knoten des Baumes Typannahmen und -voraussetzungen definiert. Diese ergeben *Constraints* für die entsprechenden TPHs. Danach können diese Constraints entweder erweitert oder verkleinert werden. Eine formale Definition hierzu ist in [Plü07, S. 8] beschrieben. Existieren am Ende der Typinferenz mehrere unterschiedliche Mengen von Constraints für einen TPH, so wird als Typ für diesen der Intersektionstyp dieser gebildet.

Die aktuelle Implementierung von Java-TX berücksichtigt zur Typinferenz ausschließlich die Typen, welche explizit mittels `import` Anweisung in den Quellcode inkludiert wurden. Dies basiert auf Laufzeitkomplexität der Typinferenz, welche mit steigender Anzahl an möglichen Typen zunimmt. Ein Beispiel für die globale Typinferenz ist in Listing 2.1 oder Listing 2.2 dargestellt. Zu beachten ist hierbei, dass die `import` Anweisungen aus Listing 2.4 in Java-TX hinzugefügt werden müssten, für eine korrekte Typinferenz. Als Ergebnis einer korrekte Typinferenz wir auf die Typen der Klassen `OLFun` in Abbildung 2.1 und Abbildung 2.3 verwiesen.

```
import java.lang.Double;
import java.lang.Integer;
import java.util.Vector;
import java.lang.Boolean;
```

Listing 2.4: `import` Anweisungen für die Klasse `OLFun`.

2.5. Theoretische Lösung zur Übersetzung echter Funktionstypen

Rückschlüsse aus Java vergleichbaren Sprachen

Dieser Abschnitt beschäftigt sich mit der theoretischen Lösung der *Type Erasure* in Java-TX für echte Funktionstypen. Durch die vorgestellte Lösung muss eine Unterscheidung der echten Funktionstypen auf Basis derer Typargumente möglich sein. Hierdurch soll die Kompilierung einer Klasse wie die Klasse `OLFun` aus Listing 2.2 ermöglicht werden. Die in Abschnitt 2.3 „Übersetzung generischer Parameter in Java vergleichbaren Sprachen“ vorgestellten Ansätze zur übersetzung generischer Parameter zeigen, dass hierzu bereits Möglichkeiten publiziert und umgesetzt wurden. In C# ist es daher möglich, eine solche Klasse zu erstellen und kompilieren. Dabei kann die Überladung direkt durch die *konstruierten generischen Klassen* aufgelöst werden. Der Code hierfür kann Listing L.4 entnommen werden. Dabei wurde das Interface `IFun1` äquivalent zu `Fun1$$` deklariert. Wie zu erkennen ist, kann dennoch keine Überladung der Methode `m` auf Basis deren Rückgabetyp erfolgen. Die Erkenntnisse aus C# können jedoch nicht für die Umsetzung der heterogenen Übersetzung echter Funktionstypen in Java-TX verwendet werden. Dazu müsste die Spezifikation der JVM angepasst werden, sodass eine *just-in-time Spezialisierung* verwendet werden kann. Im Rahmen dieser Arbeit muss jedoch ein Lösungsansatz erarbeitet werden, welcher mit der aktuellen JVM umzusetzen ist. Dies spiegelt die Abwärtskompatibilität von Java und die Intention von Java-TX wieder. Durch die Erkenntnisse aus Scala konnten unter anderem neue Möglichkeiten der JVM herausgefunden werden. So konnte die Klasse `OLFun`

aus Listing 2.3 zeigen, dass auf der JVM Methoden auf Basis derer *Rückgabetypen und* dem Unterschied in *mindestens einer Typvariablen* eines formalen Parameter überladen kann. Diese Eigenschaft kann potentiell für die Lösung der Übersetzung verwendet werden. Mit dieser erweiterten Überladung ist es möglich, Klassen wie `OLFun` aus Listing 2.3 bereits ohne Anpassung der Übersetzung selbst zu kompilieren. Die theoretische Lösung muss dennoch gewährleisten, dass die Umsetzung in Java-TX der Klasse `OLFun` mit der letzten auskommentierten Methodendeklaration kompilierbar ist. Zudem wird angenommen, dass die erweiterte Überladung in Scala keine geplante Eigenschaft ist, da dies das generelle Prinzip der *nicht-Überladung* ausschließlich im Rückgabetyp verletzt. Somit kann aus der Übersetzung von generischen Parametern in Scala nur indirekt eine Lösung für die Übersetzung dieser in Java-TX abgeleitet werden. Pizza hingegen liefert mit dessen Ansatz zur heterognen Übersetzung der generischen Parameter den Grundstein, für diese in Java-TX . [OW97] schlug vor, für jede Parametrisierung des generischen Typs einen spezifischen Typ zu generieren. Aufgrund der Vielzahl verschiedener Kombinationen schlug dieser jedoch vor, dies ausschließlich für primitive Datentypen einzusetzen. Die Referenztyp als Typargumente sollten somit dennoch mit der Type Erasure verwendet werden.

Heterogene Übersetzung echter Funktionstypen

Die Lösung zur heterogenen Übersetzung in Java-TX bedient sich dem Ansatz von [OW97] zur heterogenen Übersetzung. Sollten zwei Methodendeklarationen existieren, welche sich ausschließlich in den Typargumenten eines Funktionstyps, in deren formalen Parametern unterscheiden, so sollen diese dennoch überladen werden können. Dies soll dabei durch eine heterogene Übersetzung der Funktionstypen ermöglicht werden. Somit wird der Ansatz der heterogenen Übersetzung aus Pizza verwendet. Zudem wird indirekt die erweiterte Überladung aus Scala umgesetzt, da die heterogene Übersetzung immer einen neuen Typ einführt, sollte sich ein Typargument eines formalen Parameter unterscheiden. Dieser Ansatz wird jedoch erweitert, da durch die heterogene Übersetzung auch die letzte auskommentierte Deklaration von `m` aus Listing 2.3 kompilierbar ist. Die Definition von *überschreibungs Äquivalenz* der Java Sprachdefinition [Ora21b] muss somit angepasst werden. Nachfolgende Definitionen wurden dabei analog zu [Ora21b] mit entsprechenden Anpassungen formuliert.

Definition 2.5.1 (Funktional spezialisierte Signatur)**.** Eine Signatur $\mathtt{s}_1$ ist die *funktional spezialisierte Signatur* der Signatur $\mathtt{s}_2$, falls:

1. $\mathtt{s}_1$ und $\mathtt{s}_2$ besitzen den selben Namen, und
2. $\mathtt{s}_1$ ist die gleiche Signatur wie $\mathtt{s}_2$, wobei vor der Type Erasure die Funktionstypen heterogen übersetzt wurden.

Definition 2.5.2 (Funktionale Subsignatur)**.** Die Signatur einer Methode $\texttt{m}_1$ ist ein *funktionale Subsignatur* der Signatur einer Methode $\texttt{m}_2$, falls:

- $\texttt{m}_2$ besitzt die selbe Signatur wie $\texttt{m}_1$, oder
- die Signatur der Methode $\texttt{m}_1$ ist die selbe wie die *funktional spezialisierte Signatur* der Methode $\texttt{m}_2$.

Definition 2.5.3 (Überschreibungs Äquivalenz)**.** Zwei Methodendeklarationen $\texttt{m}_1$ und $\texttt{m}_2$ sind *überschreibungs Äquivalent* falls:

1. $\texttt{m}_1$ und $\texttt{m}_2$ besitzen die Signaturen $\texttt{s}_1$ und $\texttt{s}_2$, und
2. $\texttt{s}_1$ eine *funktionale Subsignatur* von $\texttt{s}_2$ oder $\texttt{s}_2$ ein *funktionale Subsignatur* von $\texttt{s}_1$ ist.

Diese angepasste Definition erlaubt somit die Überladung in Java, äquivalent zur Überladung, bei generischen Typen als formaler Parameter in Scala und erweitert diese sogar. Dies wird erreicht, indem die legale Deklaration in Scala aus Listing 2.3 und allgemein die Überladung in Typargumenten von Funktionen als *nicht überschreibungs äquivalent* in Java definiert wird. Trotz der erweiterten Überladung werden alle Funktionstypen heterogen übersetzt, sodass die Überladung in Java-TX noch mächtiger als diese ist. Bei der heterogenen Übersetzung beispielsweise einer Methode wird ausschließlich der Typ `Fun`N`$$` in einem formalen Parameter oder dem Rückgabetyp heterogen übersetzt. Somit handelt es sich bei dieser Lösung um eine ausschließliche Anpassung der Übersetzung und Überladung echter Funktionstypen. Dies bewirkt zwar eine möglicherweise unnötige Generierung von Spezialisierungen, falls beispielsweise die Überladung ohne Spezialisierungen bereits korrekt wäre, wie in Scala. Im Gegensatz dazu wird jedoch die Kompilierzeit verringert, indem die Auflösung der Methoden im aufrufenden Code keine zusätzliche Entscheidungslogik benötigt. Diese Entscheidungslogik müsste prüfen, ob eine Spezialisierungen benötigt wird und falls ja, diese als Signatur und Deskriptor einsetzen. Außerdem entfällt somit die Entscheidungslogik, welche bei der Bytecodegenerierung hätte verwendet werden müssen. Diese müsste überprüfen, ob eine Spezialisierung notwendig ist und falls ja, müsste eine bereits existierende Methode ohne Spezialisierung angepasst werden. Des Weiteren müsste die Logik abdecken, dass nur bei benötigen einer Spezialisierung, eine solche generiert wird. Somit wird hier der Weg der Einfachheit über den Weg der maximalen Speicherplatzoptimierung favorisiert.

Möglichkeiten zur heterogenen Übersetzung echter Funktionstypen

Nachfolgend werden die verschiedenen Möglichkeiten für die Spezialisierungen des `Fun`N`$$` Typen und deren Vor- und Nachteile in kürze erläutert. Zur Namensgebung wurden zwei

Möglichkeiten identifiziert. So kann die Spezialisierung die Form `Fun`N`$$`M annehmen, wobei M für den Index der Variante, beginnend ab eins, verwendet wird. Das Basisinterface erhält somit inoffiziell den Index null, wobei dieser nicht ausdrücklich im Typ selbst auftaucht. Ein Vorteil dieser Namensgebung wäre die potentielle Wiederverwendbarkeit der Spezialisierungen, sollten diese selbst generische Parameter besitzen. Dies beschreibt den Umstand, sollte eine Spezialisierung analog zum Basistypen `Fun`N`$$` benötigt werden. Nachteile dieser Namensgebung wäre jedoch die Intransparenz dieser, da bei unterschiedlichen Kompilierungen möglicherweise unterschiedliche Namen vergeben werden. Dies würde zum einen zu einer möglichen Inkompatibilität mit weiterem Bytecode führen und zum anderen zu einem erhöhtem Aufwand bei der Kompilierung. Dieser höhere Aufwand ist mit dem Abspeichern der Zuordnungen zwischen Index und Typargumenten zu begründen.
Eine weitere Namensgebung besteht darin, den Namen dynamisch auf Basis der Typargumente zu vergeben. So besitzt jede Spezialisierung den „Namensstamm“ `Fun`N`$$`. Hiernach folgt eine Sequenz der Namen der Typargumente. Als Name der Typargumente wird der Deskriptor verwendet, um gleichnamige Klassen in unterschiedlichen Paketen oder gleiche Typvariablen abzubilden. Bei nicht aufgelösten TPHs wird die Zeichenfolge `LTPH;` anstelle des Deskriptors verwendet. Da das Zeichen `/` jedoch nicht zulässig in Klassennamen ist, wird dies durch das Zeichen `$` ersetzt. Hierbei wird angenommen, dass dies wahrscheinlich zu keinen Uneindeutigkeiten zwischen Deskriptoren zweier Typargumente führt. Dies Annahme wird getroffen, da Java selbst diese Substitution von Namen innerer Klassen hin zu deren Klassennamen verwendet. Die einzelnen Typargumente werden durch die Sequenz `$_$` getrennt. Somit erfolgt eine Substitution der Form: `;` $\rightarrow$ `$_$` in dem Deskriptor. Diese Substitution wurde ausgewählt, da das Zeichen `$` für generierten Code und `_` bereits als Bezeichner nach [Ora21b] definiert wurden. Somit wird diese Sequenz als unwahrscheinlich auftauchend in Typargumenten betrachtet und trennt dadurch eindeutig die Typargumente. Es wurden außerdem bewusst die Deskriptoren und nicht die Signaturen der Typargumente verwendet, da diese wiederum mit der ursprünglichen Type Erasure übereinstimmen. Der Vorteil dieser Namensgebung besteht in der ausschließlich abhängigen Generierung zu den Typargumenten und keiner weiteren Speicherung von Zuordnungen. Zudem weißt diese Namensgebung eine potentielle Eindeutigkeit auf. Nachteile dieser sind jedoch die Länge der Namen und die dennoch mögliche, wenn auch unwahrscheinliche, Uneindeutigkeit der Namen.

Außerhalb der Namensgebung existieren zudem drei Möglichkeiten zur Umsetzung der Deklaration der Spezialisierungen. Zum einen kann die Spezialisierung selbst das Basisinterface `FunN$$` erweitern und bei der Erweiterung die Typargumente spezifizieren. Um die Eigenschaft eines funktionalen Interfaces zu wahren, darf die Spezialisierung keine weitere Methode deklarieren. Ein Vorteil dieser Methode ist die mögliche Erweiterbarkeit auf beliebige generische Typen und der geringe zusätzliche Speicherplatzverbrauch. Es

wird weniger Speicherplatz verbraucht, wie wenn jede Spezialisierung eine eigene Methodendeklaration beinhaltet. Ein Nachteil konnte bei dieser Spezialisierung bislang nicht erkannt werden, da diese zudem keine Bridging-Methoden generiert.
Eine weitere Art der Spezialisierung wäre die Generierung eines äquivalenten Interfaces zu `Fun`N`$$`, welches ebenfalls generisch wäre. Dieses würde sich jedoch im Namen unterscheiden. Außerdem würde jedes Interface eine eigene Methode `apply` deklarieren. Vorteile für diese Umsetzung konnten bislang nicht gefunden werden. Ein Nachteil ist jedoch der hohe Speicherplatzverbrauch durch die redundante Deklaration der Methode `apply`.
Der letzte Ansatz wäre ähnlich zu dem vorherigen, nur das die Spezialisierung keine Typparameter besitzt. Da es sich ausschließlich um ein sehr spezifisches Interface handelt, bergen die Typparameter keine Vorteile. Somit würde die Methode `apply` direkt die korrekten Typannotationen erhalten. Als Vorteil kann gegenüber der vorherigen Methode definiert werden, dass diese keine Umwandlung der Typparameter bei der Type Erasure unterliegt. Somit müssen keine Typumwandlung zur Laufzeit durchgeführt werden. Ein Nachteil ist jedoch erneut der erhöhte Speicherplatzverbrauch durch die annähernd redundanten Deklarationen.

Finale heterogene Übersetzung von `Fun`N`$$`

Nachfolgend wird die finale Umsetzung der heterogenen Übersetzung beschrieben. Basierend auf den Vor- und Nachteilen aller Ansätze und unter Berücksichtigung der Erweiterbarkeit auf beliebige parametrisierte Typen, werden die folgenden Ansätze kombiniert: Der Name der Spezialisierung besteht aus dem Namensstamm `Fun`N`$$` und anschließend den Typargumenten. Die Typargumente werden durch deren Deskriptoren dargestellt und diese dem Namensstamm hinzugefügt. TPHs welche nicht aufgelöst werden konnten, werden anstelle als `LTPH;` dem Namensstamm hinzugefügt. Außerdem werden diese analog zu sonstigen Typvariablen als generischer Parameter des Interfaces deklariert. Somit besitzt anschließend das spezialisierte Interface Typparameter, bestehend aus den TPHs und den Typvariablen, welche in den Typargumenten übergeben wurden. Vor Hinzufügen der Deskriptoren zum Namensstamm, werden diese den folgenden Substitutionen unterzogen:

1. `.` → `/`
2. `/` → `$`
3. `;` → `$_$`

Das spezialisierte Interface erweitert dabei das Basisinterface `Fun`N`$$` und spezifiziert dabei dessen Typparameter. Dies hat zur Folge, dass Spezialisierungen mit geringem

Speicherplatzverbrauch generiert werden. Durch diese heterogene Übersetzung wird sichergestellt, dass jede Kombination von Typargumenten eindeutig einem spezialisierten Interface zuordenbar ist. Ein Beispiel der finalen heterogenen Übersetzung ist in Listing 2.5 dargestellt.

```
public interface Fun1$$<T1,R> {
    public R apply (T1 arg1);
}

public interface
    Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
     extends Fun1$$<Integer, Integer> {}

public interface
    Fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$
     extends Fun1$$<Double, Double> {}
```

Listing 2.5: Finale heterogene Übersetzung echter Funktionstypen.

3. Umsetzung

3.1. Prototyping

Zur Bytecodegenerierung in Java-TX wird die Bibliothek `org.objectweb.asm` von [Bru21] verwendet. Bei ASM handelt es sich nach [Bru21] um eine Bibliothek zur Manipulation und Analyse von Bytecode. Dadurch können sowohl existierende Klassen geändert, als auch zur Laufzeit neue Klassen generiert werden. Dabei werden diese direkt als Bytecode generiert bzw. manipuliert. Ein Vorteil von ASM liegt laut [Bru21] darin, dass dieses mit dem Fokus auf der Performanz entwickelt wurde. Somit wurde ASM so klein und schnell wie möglich implementiert. Einsatzgebiete findet die Bibliothek dabei sowohl in dynamischen, als auch in statischen Systemen. Aus diesen Gründen eignet sich ASM für die Bytecodegenerierung in Java-TX . Referenzprojekte werden in [Bru21] mit den Compilern von Groovy und Kotlin genannt. Die Bibliothek selbst steht dabei direkt als `jar`-Datei zum Download bereit. Außerdem kann diese in Building-Tools wie Maven oder Gradle integriert werden. Die aktuellste Version stellt dabei zum Stand dieser Arbeit ASM-9.2 dar, welche im Juni 2021 veröffentlicht wurde [Bru21]. Zudem bietet IntelliJ verschiedene Plugins für ASM an. Hierdurch wird unter anderem die Möglichkeit geschaffen, den ASM Code für eine Java-Sourcedatei direkt in IntelliJ anzuzeigen. Es wird dabei eine gleichnamige Klasse generiert, welche den Zusatz `Dump` besitzt. Diese Klasse umfasst zudem eine statische Methode namens `dump`. Mithilfe dieser Methode kann anschließend ein Byte-Array generiert werden, welches den Bytecode der zu generierenden Klasse umfasst. Diese einfache und schnelle Möglichkeit, ASM Code zu generieren, kann zur Erstellung eines Prototyps genutzt werden. Ziel des Prototyps ist dabei, die heterogene Übersetzung von Funktionstypen in Java zu verifizieren. Die Namensgebung und Kompilierung der zu erstellenden Klassen kann dabei durch manuelle Generierung der Sourcecode Dateien erfolgen. Diese wiederum können genutzt werden, um mithilfe des IntelliJ Plugins den entsprechenden ASM Code zu generieren.

In einem ersten Schritt wurde die Überladung aus Scala in Java überprüft. Dabei konnte festgestellt werden, dass der entsprechende Sourcecode nicht von `javac` kompiliert werden konnte. AMS kann jedoch korrekten und inkorrekten Sourcecode in Bytecode umwandeln. So konnte der inkorrekte Sourcecode durch das ASM Plugin in IntelliJ manuell in ASM Code übersetzt werden. Anschließend mussten die entsprechenden Teilstücke des ASM Codes zusammengesetzt werden. Hierdurch konnte der eigentlich falsche Bytecode generiert

werden. Zum Abschluss musste überprüft werden, ob der Bytecode korrekt bezüglich der JVM ist. Das bedeutet, ob dieser auf der JVM ausführbar ist. Dies konnte durch die Verifikation mithilfe der JVM selbst erfolgen, indem der Bytecode entsprechend geladen und ausgeführt wurde. Dabei konnte festgestellt werden, dass der Bytecode korrekt ist. Somit kann zwar der Bytecode durch ASM generiert werden, birgt dennoch den Nachteil, dass die heterogene Übersetzung damit nicht vollständig umsetzbar ist. Zusammengefasst wird daher angenommen, dass es sich um einen Seiteneffekt oder Fehler in der JVM handelt. Entsprechende Verhalten sollte laut Definition nicht möglich sein.

Anschließend wurde überprüft, ob die finale heterogene Übersetzung mittels ASM generiert werden kann. Dazu wurden manuell die entsprechenden Sourcecode Dateien erstellt und in ASM Code überführt. Dabei entstanden die in Abschnitt D „ASM Code des Prototypen“ dargestellten Sourcecode Dateien. Diese umfassen dabei die `Dump` Klassen, welche entsprechend den Bytecode durch die Methode `dump` generieren. Verwendet werden hierzu hauptsächlich die Klassen `ClassWriter` und `MethodVisitor`. Die Klasse `ClassWriter` bietet die Möglichkeit, mittels der Methode `visit` die Klasse selbst zu spezifizieren. So kann dabei die Java Version, Annotationen, der Name der Klasse oder auch deren Superklasse oder Interfaces definiert werden. Zum Abschluss kann die Generierung mithilfe der Methoden `visitEnd` und `toByteArray` abgeschlossen werden. Ergänzend hierzu bietet die Klasse `MethodVisitor` die Methode `visitMethod` an. Diese muss vor dem Abschluss der Generierung verwendet werden. Dabei kann durch diese Methode der Name der zu erstellenden Methode, deren Annotationen, der Deskriptor und die Signatur spezifiziert werden. Die korrekte Generierung konnte anschließend durch die Verwendung der entsprechenden Klassen in einer weiteren verifiziert werden. Somit konnte im Rahmen des Prototyping gezeigt werden, dass die finale Lösung zur heterogenen Übersetzung durch ASM generiert werden kann. Im Kapitel 4 „Implementierung“ muss folgend die Implementierung der Übersetzung in Java-TX beschrieben werde.

3.2. Aktueller Stand von Java-TX

Anschließend wird der aktuelle Stand des Projekts Java-TX kurz dargestellt. Ein Compiler allgemein kann nach [Aho08, S. 6] in verschieden Phasen untergliedert werden. Diese sind in Abbildung 3.1 dargestellt. So beginnt dieser mit der Lexikalischen Analyse, über die Syntax- und Semantische Analyse bis hin zur Codegenerierung. Dabei können bei der Codegenerierung wiederum der Zwischencode- und der Codegenerator selbst unterschieden werden. Nach jeweils den beiden Phasen kann eine Codeoptimierung stattfinden. Zu beachten ist hierbei, dass Java ausschließlich ein einziges Mal Code generiert. Dieser Code, der Bytecode, ist dabei maschinenunabhängig. Somit entfallen die letzten beiden

Phasen aus Abbildung 3.1. Die Analysen können nach [Aho08, S. 7] als sogenanntes *Front-End* des Compilers zusammengefasst werden. Die Synthese hingegen, welche aus der Zwischendarstellung und einer Symboltabelle den Code generiert, wird als *Back-End* des Compilers bezeichnet. In der Praxis können laut [Aho08, S. 7] verschiedene Phasen kombiniert werden oder wie in Java entfallen.

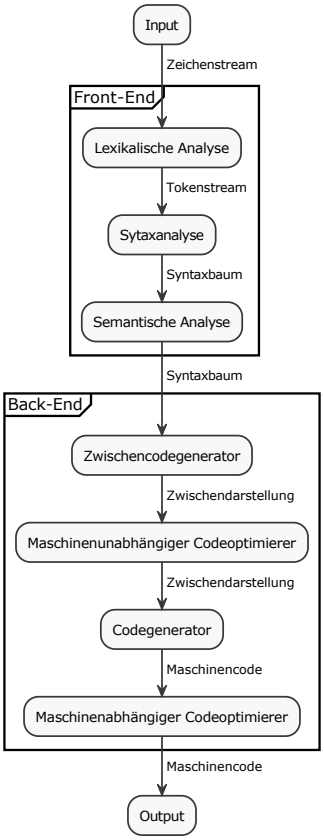


Abbildung 3.1.: Phasen eines Compilers nach [Aho08, S. 6].

Java-TX kann auch entsprechend in Front-End und Back-End untergliedert werden. Abbildung 3.2 stellt dabei die Idealdarstellung der Komponenten entsprechend des Front- und Back-Ends dar. Wie zu erkennen ist, spiegeln sich die einzelnen Phasen des Front-Ends in entsprechenden Komponenten wieder. Die zusätzlichen Funktionalitäten von Java-TX wie die Typinferenz oder Typunifikation werden dabei ebenfalls als Komponenten des Front-Ends aufgefasst. Diese Zuordnung wird vorgenommen, da diese beiden Funktionalitäten unabdingbar für die Generierung der Zwischendarstellung, in diesem Fall der Abstract Syntax Tree (AST), sind. Zur Auflösung der TPHs entsprechend der Unifikation wird außerdem die Komponente des Resolver benötigt. Mithilfe dessen kann idealerweise der Bytecodegenerierung ein AST mit bereits aufgelösten TPHs übergeben werden. Die Bytecodegenerierung und Dateigenerierung befinden sich dabei entsprechend der Phasen von Abbildung 3.1 im Back-End.

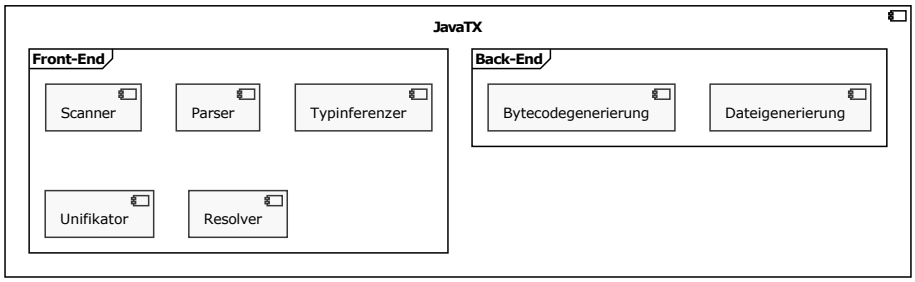


Abbildung 3.2.: Ideal-Zustand der Komponenten von Java-TX .

Diese idealisierte Darstellung ist aktuell in Java-TX nicht umgesetzt. Eine schematische Darstellung der aktuellen Komponenten ist in Abbildung 3.3 dargestellt. Wie hier zu erkennen ist, unterschiedet sich das Front-End kaum von der idealen Aufteilung. Einzig die Komponente des Resolvers ist im Front-End nicht vorhanden. In Abbildung 3.3 ist zu erkennen, dass kein klar definiertes Back-End in Java-TX existiert. Genauer kann zusammengefasst werden, dass das Back-End einzig aus der Komponente `Bytecodegenerierung` besteht. Diese wiederum setzt sich aktuell aus dem Resolver, der Bytecodeerstellung und der Dateierstellung zusammen. Somit liegt hier keine „Single Responsibility“ dieser Komponente vor, sondern sie vereint das gesamte Back-End. Zur Folge hat dies, dass in den Klassen zur Bytecodegenerierung sowohl die Erstellung des Bytecodes, als auch die gleichzeitige Auflösung der TPHs stattfindet. Zudem wurden Methoden deklariert, welche zur Dateierstellung und -manipulation verwendet werden. Resultat hiervon ist sowohl verworrener Code, als auch die Definition langer Methoden. Außerdem wurden häufig (globale) Zustände in Methoden der Bytecodegenerierung manipuliert, anstelle der Verwendung von Argumenten und Rückgabewerten. Alles in allem führt dies aktuell zu einer zeitintensiven Einarbeitung und schweren Formulierung von Diagrammen, welche das Verhalten korrekt beschreiben. Ein weiteres Problem ist die nicht-Implementierung von automatisch generierten Methoden, welche von Subklassen korrekt implementiert werden müssten. So existieren in Java-TX einige Methoden, welche dadurch semantisch inkorrekt sind. Des Weiteren wurden in Java-TX uneindeutige Schnittstellen definiert. Die Klasse `TypeToSignature` beispielsweise generiert die korrekte Signatur für einen Typ des AST. Hingegen die Klasse `TypeToDescriptor` generiert inkorrekte Deskriptoren. Deskriptoren beginnen mit dem Zeichen `L` und enden mit dem Zeichen `;`. `TypeToDescriptor` hingegen generiert Deskriptoren ohne diese Zeichen. Somit muss nach jedem Aufruf der Deskriptor um diese Zeichen ergänzt werden.

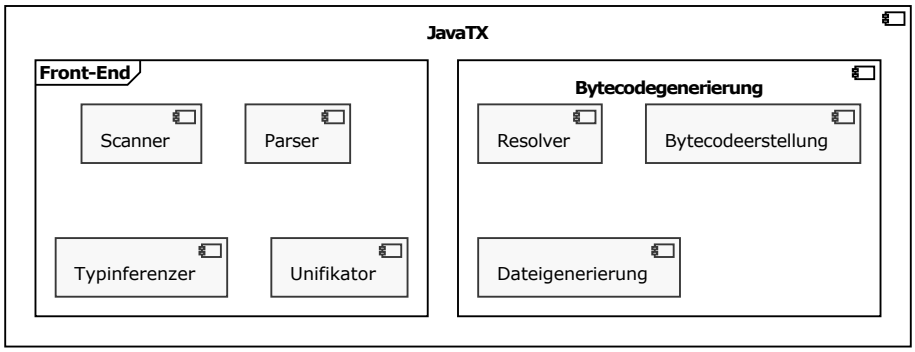


Abbildung 3.3.: Ist-Zustand der Komponenten von Java-TX .

Für die Bytecodegenerierung wurde in Java-TX eine Test-Suite angelegt. Diese besteht aus Ende-zu-Ende Tests. Es wurden `.jav`-Dateien erstellt, welche anschließend in den Tests kompiliert werden. Der Bytecode wird anschließend mithilfe des `URLClassLoader` geladen. Anschließend wird eine Instanz der kompilierten Klasse erstellt. Diese Tests birgt dabei einige Probleme:

Keine Schnittstellentests führen zur direkten Abhängigkeit der Test zu allen weiteren Komponenten.

Mangelnde Testfälle führen zu unentdeckten Fehlern.

Instanziierung der Klassen ist nicht gleichbedeutend zu semantisch korrektem Bytecode. Diese Annahme konnte manuell verifiziert werden.

3.3. Möglichkeiten zur Umsetzung

Für eine nachhaltige Softwareentwicklung müsste die Bytecodegenerierung zu Beginn refactored werden. So muss diese vom Zustand aus Abbildung 3.3 in den Zustand aus Abbildung 3.2 überführt werden. Aufgrund des hohen Aufwands in Relation zur Verfügung stehenden Zeit dieser Arbeit, wird hiervon abgesehen. Zudem würde dies weit mehr Arbeit beanspruchen, als die Implementierung der heterogenen Übersetzung echter Funktionstypen. Ein entsprechendes Refactoring muss jedoch in Zukunft durchgeführt werden. Aus diesem Grund müssen die in diesem Abschnitt dargestellten Umsetzungsmöglichkeiten einfach zu refactorn sein bzw. kein Refactoring benötigen.

Unifikation heterogener Typen

Die erste Möglichkeit wäre eine Anpassung der Unifikation. So könnte der Unifikations-Algorithmus angepasst werden, sodass dieser direkt die heterogenen Übersetzungen echter Funktionstypen bestimmt. Anschließend müsste die Bytecodegenerierung sicherstellen, dass die entsprechenden heterogenen Funktionstypen erstellt werden. Diese Umsetzung würde somit einzig diese beiden Komponenten beeinflussen. Zur Folge hätte dies jedoch, dass der Unifikation weitere Regeln hinzugefügt werden müssten. Das Ergebnis der Unifikation würde somit erheblich vergrößert werden, da für jede heterogene Übersetzung ein entsprechender Constraint vorhanden wäre. Zudem würde ein allgemeiner Unifikator Umsetzungsdetails der heterogenen Übersetzung aufweisen. Somit wäre die „Single Responsibility“ der Unifikation verletzt. Dies hätte zur Folge, dass bei Anpassung der heterogenen Übersetzung der Unifikator angepasst werden müsste. Sollte somit zukünftig beispielsweise die JVM Unterstützung für heterogene Übersetzungen anbieten, müsste eine derart zentrale und komplexe Komponente angepasst werden. Eine solch erforderte Änderbarkeit an einer derart zentralen Komponente wäre aus Sicht des Autors deplatziert.

Substitution im AST

Eine weitere Möglichkeit zur Umsetzung wäre das Hinzufügen einer Substitutionskomponente. Dieser wird der AST mit den homogenen Funktionstypen übergeben. Anschließend werden die Funktionstypen durch deren heterogene Übersetzung im AST substituiert. Hierzu muss der Resolver verwendet werden, welcher die TPHs im AST auflöst. Hiernach müssten die korrekten heterogenen Übersetzungen abgeleitet werden. Der Bytecodegenerierung wird anschließend der korrekte AST übergeben. Erneut muss in der Bytecodegenerierung sichergestellt werden, dass die entsprechenden heterogenen Übersetzungen generiert werden.

Diese Umsetzung erfordert die Definition klarer Schnittstellen. So muss der Substitutionskomponente ein klar definierter AST übergeben werden. Anschließend muss eine Schnittstelle zur Bytecodegenerierung verwendet werden, welche den substituierten AST entgegennimmt. Eine solch klare Trennung ist wie in Abbildung 3.3 dargestellt nicht gegeben. Das Refactoring hin zu entsprechenden Schnittstellen würde wie eingangs beschrieben in keinem Verhältnis zur eigentlichen Umsetzung stehen. Somit ist diese Möglichkeit für die aktuelle Umsetzung ungeeignet. Nach einem entsprechenden Refactoring sollte jedoch diese Möglichkeit erneut evaluiert werden.

Auflösung korrekter Typen

Die Umsetzung der heterogenen Übersetzung kann außerdem im Resolver erfolgen. Dabei kann eine Funktionalität implementiert werden, welche die heterogenen Funktionstypen anstelle der homogenen auflöst und somit zurückliefert. Somit kann man anschaulich von einem „Einklingen“ in die Funktionalität der Typauflösung sprechen. Wie auch bei den vorherigen Implementierungsmöglichkeiten muss auch bei diesem Ansatz sichergestellt werden, dass die heterogenen Funktionstypen generiert werden. Diese Möglichkeit würde sich einzig eine Anpassung der Komponenten des Resolvers und der Bytecodeerstellung beinhalten.

Wie auch bei dem Ansatz der Unifikation würde somit eine allgemeine Komponenten mit Umsetzungsdetails der heterogenen Übersetzung versehen werden. Dadurch würde erneut die „Single Responsibility“ dieser Komponente verletzt werden. Erneut müsste bei Änderungen der heterogenen Übersetzung der Resolver angepasst werden. Dies widerstrebt dabei dem Gedanken, dass es sich beim Resolver um eine Komponente handelt, welche eine klar definierte Schnittstelle zur Typauflösung bereitstellt.

Anpassung der Bytecodegenerierung

Eine letzte Umsetzungsmöglichkeit besteht in ausschließlich der Anpassung der Bytecodeerstellung. Es handelt sich somit um eine Änderung an einer zentralen Komponente. Die Bytecodeerstellung muss dabei erneut sicherstellen, dass die heterogenen Funktionstypen erstellt werden. Dies kann realisiert werden, indem bei Verwendung eines homogenen Funktionstyps der entsprechende heterogene abgeleitet wird. Anschließend wird überprüft, ob der Bytecode des entsprechenden heterogenen Funktionstyps bereits existiert. Sollte dieser bereits existieren, so wird keine Aktion durchgeführt. Falls der Bytecode noch nicht existiert, so wird dieser entsprechend generiert und in der entsprechende `.class`-Datei abgespeichert.
Dieser Ansatz erfordert zudem in der Bytecodeerstellung eine Anpassung der Generierung von Signaturen und Deskriptoren. So müssen die entsprechenden Funktionalitäten derart angepasst werden, dass auch diese anstelle der homogenen Funktionstypen die heterogenen verwenden. Entsprechend müssen diese filtern, wann ein homogener Typ vorliegt, entsprechend den heterogenen bestimmen und anschließend diesen in der Signatur/ dem Deskriptor verwenden.

Es handelt sich somit um eine kleine und zentrale Änderung an Funktionalität. Zudem entspricht dieser Ansatz dem Grundgedanken, dass in Java bereits generische Typen existieren und diese weiter verwendet werden sollen. Einzig deren Umsetzung im Bytecode, soll

von eine homogenen zu einer heterogenen Variante geändert werden. Somit handelt es sich nicht um eine Änderung im Typsystem oder ähnlichem, sondern um ein Umsetzungsdetails der Typen im Bytecode. Dieser Ansatz verletzt somit nicht wie andere das Prinzip der „Single Responsibility“. Zukünftige Änderungen in der Übersetzung müssen zentral bei der Bytecodeerstellung erfolgen. Dies ist dabei die korrekte Kapselung der Funktionalität.

Umzusetzender Ansatz

Für die Umsetzung wurde die Anpassung ausschließlich der Bytecodegenerierung ausgewählt. Wie beschrieben handelt es sich um die korrekte Kapselung der entsprechenden Funktionalität. Dadurch wird in keiner Komponente das Prinzip der „Single Responsibility“ verletzt. Außerdem erfolgen zukünftige Änderungen zentral an nur einer Komponente. Bei zukünftigem Refactoring kann zudem die Funktionalität entsprechend beibehalten werden. Selbst bei Änderung der Schnittstellen zur Bytecodegenerierung sollte die Funktionalität entsprechend korrekt sein. Zudem ist kein Refactoring vor der Umsetzung nötig. Der Ansatz kann direkt in den aktuellen Code implementiert werden.

4. Implementierung

Nachdem im Abschnitt 3.3 „Möglichkeiten zur Umsetzung“ diverse Umsetzungsmöglichkeiten beschrieben wurde, wird in diesem Kapitel die Implementierung des ausgewählten Ansatzes näher beleuchtet. Der umzusetzende Ansatz ist dabei dieser, welcher ausschließlich eine Anpassung der Bytecodegenerierung erfordert. Eine Skizze der Komponenten von Java-TX und somit der Bytecodegenerierung wurde dazu bereits in Abbildung 3.3 dargestellt. Zum Verständnis der Implementierung müssen eingangs relevante Klassen der Bytecodegenerierung eingeführt werden. Darunter fallen beispielsweise die Klassen, welche im AST die Typen repräsentieren. Außerdem die Klassen zur Generierung von Signaturen und Deskriptoren und die Klasse zur Bytecodeerstellung.

4.1. Abstract Syntax Tree-Typen

Die Typen des AST wurden im Package `syntaxtree.type` definiert. Ein Klassendiagramm dieser ist in Abbildung 4.1 abgebildet.

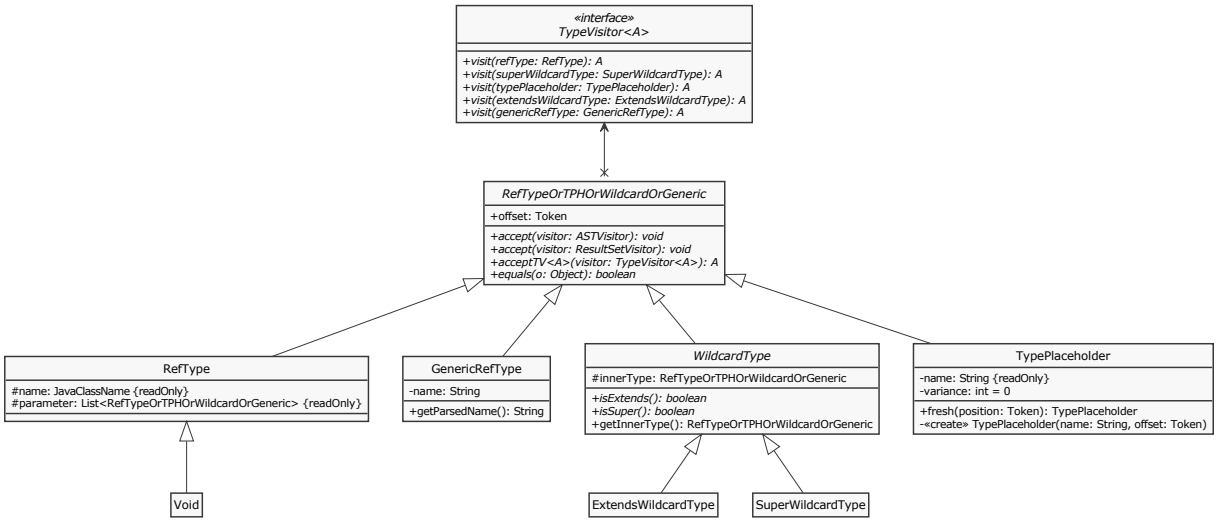


Abbildung 4.1.: AST-Typen in Java-TX .

Die abstrakte Klasse `RefTypeOrTPHOrWildcardOrGeneric` bildet dabei die oberste Hierarchieebene der AST-Typen. Diese deklariert dabei vor allem die abstrakten `accept`-Methoden. Die Methode `acceptTV` ist hierbei von besonderer Bedeutung für die Bytecodegenerierung. Wie in der Deklaration des Interfaces `TypeVisitor` zu erkennen ist, deklariert

dieses mehrere `visit`-Methode. Jede `visit`-Methode besitzt dabei eine nicht-abstrakten Spezialisierungen von `RefTypeOrTPHOrWildcardOrGeneric` als formalen Parameter. Zudem handelt es sich bei `TypeVisitor` um ein generisches Interface mit dem Typparameter `A`. Dieser stellt dabei den Rückgabetyp aller `visit`-Methoden dar. `TypeVisitor` ist somit eine Implementierung des sogenannten *visitor Pattern* für die Typhierarchie von `RefTypeOrTPHOrWildcardOrGeneric`. Das visitor Pattern wird verwendet um eine zentrale Implementierung von bestimmten Funktionalitäten für diese Hierarchie bereit zu stellen. Als zentralen Implementierung wird hierbei die jeweilige Realisierung einer solchen Funktionalität in einer separaten Klasse verstanden. Zur Verwendung einer Funktionalität muss ausschließlich die `acceptTV`-Methode mit der entsprechenden Implementierung aufgerufen werden.

Die Semantik der Spezialisierungen dieser Typhierachie ist die folgende:

`RefType` repräsentiert Referenztypen in Java-TX . Diese können hierbei eine Liste von Typparametern besitzen. Ist die Kardinalität dieser Liste größer null, so handelt es sich bei diesem Referenztyp um einen generischen Typ. Beispiele hierfür wären die Klassen `Object` oder `List<String>`.

`Void` stellt als Spezialisierung von `RefType` den speziellen Typ `void` in Java-TX dar.

`GenericRefType` ist in Java-TX eine Typvariable bzw. ein Typparameter. Dieser besitzt dabei ausschließlich einen Bezeichner und keine obere Grenze. Ein Beispiel hierfür wäre eine Klassendeklaration wie `MyClass<T>`. Nicht abbildbar ist eine Deklaration wie die folgende: `MyClass<T extends String>`. `GenericRefType` würde in beiden Deklarationen die Typvariable `T` darstellen.

`ExtendsWildcardType` repräsentiert dem Namen gemäß eine *Extends-Wildcard*.

`SuperWildcardType` repräsentiert hingegen eine *Super-Wildcard*.

`TypePlaceholder` stellen während der Kompilation die TPHs dar. Diese müssen hierbei über eine *factory*-Methode generiert werden. Bei der Bytecodeerstellung werden nicht aufgelöste `TypePlaceholder` auf einen `GenericRefType` abgebildet. Dabei erhält dieser den Namen des `TypePlaceholder` mit angefügtem `$`.

4.2. Generierung von Signaturen und Deskriptoren

Für die Generierung der Signaturen und Deskriptoren der Typhierarchie aus Abbildung 4.1 wird das visitor Pattern verwendet. Hierzu existieren bereits zwei Implementierungen. Diese Implementierungen stellen die Klassen `TypeToSignature` und `TypeToDescriptor`

dar. Ein Klassendiagramm dieser ist in Abbildung 4.2 dargestellt. Beide Implementierungen spezialisieren hierbei `TypeVisitor`. Der generische Parameter `A` wird bei der Implementierung an den Typ `String` als Typargument gebunden. Somit werden bei der Umwandlung der Typen in die Signatur bzw. den Deskriptor diese als `String` zurückgeliefert.

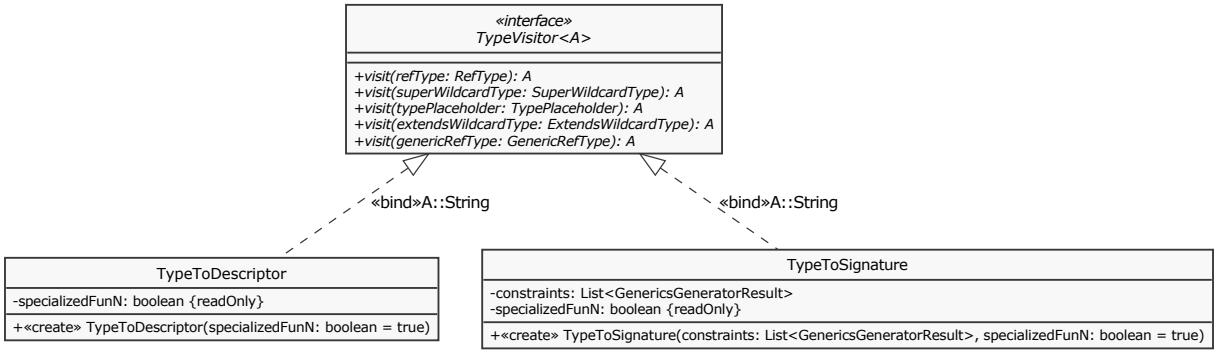


Abbildung 4.2.: Implementierungen von `TypeVisitor` zur Generierung von Signaturen und Deskriptoren.

Zur Implementierung der heterogenen Übersetzung mussten diese beiden Klassen angepasst werden. Für die Anpassung ist es von Relevanz zu erwähnen, dass echte Funktionstypen als Referenztyp betrachtet werden. Somit muss ausschließlich die `visit`-Methode für den formalen Parameter `RefType` angepasst werden. Zur Unterscheidung zwischen *normalen* Referenztypen und echten Funktionstypen wurde ein regulärer Ausdruck verwendet. Dadurch kann eine separate Behandlung für echte Funktionstypen implementiert werden. Ein Problem dabei ist jedoch, dass dieser eigenständig keine Unterscheidung zwischen den spezialisierten Funktionstypen und dem generalisierten treffen kann. Der Basisfall ist hierbei, dass jeder echte Funktionstyp in einen spezialisierter Funktionstyp umgewandelt werden muss. In Ausnahmefällen muss jedoch der Funktionstyp als der generalisierte Funktionstyp behandelt werden. Bei diesem unterscheidet sich dabei die Generierung der Signatur bzw. des Deskriptors nicht von *normalen* Referenztypen. Das Beispiel dieses Sonderfalles ist die Generierung des Bytecodes für die generalisierten Funktionstypen. Diese benötigen dabei entsprechend eine generalisierte Signatur bzw. einen Deskriptor.

Zur Umsetzung dieser Unterscheidung wurde beiden Implementierungen ein interner Zustand hinzugefügt. Dies geschieht in Form eines boolschen Attributs (`specializedFunN`), welches Standardmäßig den Wert `true` erhält. Bei diesem Zustand handelt es sich um einen finalen Zustand, welcher nach der Instanziierung nicht geändert werden kann. Der Standardfall kann durch die Konstruktoren und der dortigen Übergabe eines boolschen Wertes verändert werden. Zur Generierung einer spezialisierten Signatur bzw. Deskriptor wird nun der reguläre Ausdruck und der interne Zustand berücksichtigt. Die spezialisierten Varianten werden dabei ausschließlich generiert, sollte der reguläre Ausdruck zutreffen *und* der interne Zustand den Wert `true` besitzen.

4.3. BytecodeGenMethod

Die Klasse `BytecodeGenMethod` wurde erstellt um Bytecode zu generieren. Dabei implementiert diese erneut das visitor Pattern. In diesem Fall wird das visitor Pattern durch das Interface `StatementVisitor` deklariert. Das Pattern wird dabei nicht auf die Hierarchie von `RefTypeOrTPHOrWildcardOrGeneric`, sondern auf die der abstrakten Basisklasse `SyntaxTreeNode` definiert. Aufgrund des großen Umfangs dieser Hierarchie und der eigentlichen Irrelevanz dieser zum Verständnis der Bytecodeerstellung, wird darauf verzichtet weiter auf diese einzugehen. Relevant für die heterogene Übersetzung sind zwei Deklarationen der `visit`-Methode. Zum einen ist es diejenige, welche den Typ `LambdaExpression`, zum anderen diejenige welche den Typ `MethodCall` als formalen Parameter deklariert. Die Methode mit dem Typ `LambdaExpression` als formalen Parameter wird immer dann aufgerufen, wenn ein Lambda Ausdruck deklariert wird. Hingegen wird die Methode mit dem formalen Parameter `MethodCall` für die Erstellung des Bytecodes jeglichen Methodenaufrufs verwendet. Eine Relevanz ausschließlich dieser beiden Methoden wurde definiert, indem bei der aktuellen Implementierung ausschließlich an diesen Stellen homogene Funktionstypen generiert werden. Im nachfolgenden wird darauf verzichtet auf die genaue Implementierung dieser Methoden ein zu gehen. Ausschließlich die Änderungen und die hierfür relevanten Stellen werden genauer beschrieben.

Die Klasse `LambdaExpression` implementiert das Interface `TypeScope` und spezialisiert die Klasse `Expression`. Erst durch die Spezialisierung der Klasse `Expression` wird `LambdaExpression` als formaler Parameter einer `visit`-Methode von `StatementVisitor` relevant. Von Relevanz für die Implementierung dieser `visit`-Methode ist das Attribut `params` von `LambdaExpression`. Außerdem ist die zu implementierende Methode `getReturnType` des Interfaces `TypeScope` von gleicher Relevanz. Das Klassendiagramm zu `LambdaExpression` ist in Abbildung 4.3 dargestellt.

Für die heterogene Übersetzung werden in der `visit`-Methode von `BytecodeGenMethod` zuerst die Typargumente und der Rückgabetyp des Lambda Ausdrucks korrekt aufgelöst. Hierzu wird die Klasse `Resolver` verwendet, welcher die TPHs auflösen kann. Die Typargumente müssen hierbei zuerst von dem Typ `ParameterList` in eine Liste von `FormalParameter` überführt werden. Anschließend müssen diese in einen `Stream` von `RefTypeOrTPHOrWildcardOrGeneric` umgewandelt werden. Durch erneutes Mapping dieser Typen mit dem `Resolver` kann anschließend einer Liste aufgelöster Typargumente generiert werden. Mithilfe der aufgelösten Typen für die Rückgabe und der Typargumente, kann anschließend der Bytecode mithilfe des Interfaces `FunNUtilities` generiert werden. Im Gegensatz zu diesem Vorgehen wurde bei der homogenen Übersetzung ausschließlich auf Basis der Kardinalität der Typargumente (`params`) ein Funktionstyp erstellt.

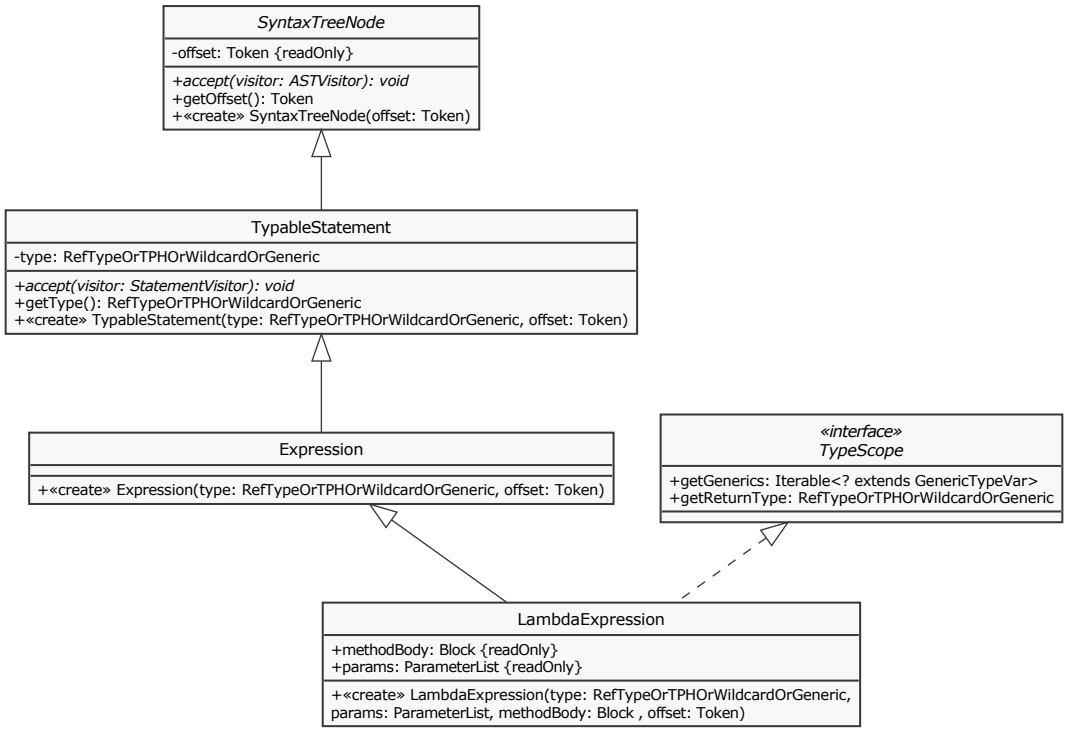


Abbildung 4.3.: Klassendiagramm der Klasse `LambdaExpression`.

Die Klasse `MethodCall` hingegen spezialisiert ausschließlich die Klasse `Statement`. Dabei spezialisiert `Statement` wiederum die Klasse `Expression`. Das dazugehörige Klassendiagramm ist in Abbildung 4.4 dargestellt. Für die Implementierung wichtig sind die Attribute `arglist` und das geerbte Attribut `type`. Ein weiteres möglicherweise relevantes Attribut ist `argTypes`. Während der Implementierung konnte jedoch festgestellt werden, dass die darin enthaltenen Typen nicht immer zu denen in `arglist` korrespondieren. Aus diesem Grund wird das Attribut `argTypes` vernachlässigt. Die Argumenttypen werden bei der heterogenen Übersetzung aus diesem Grund durch eine `ArgumentList` bzw. deren Methode `getArguments` bezogen. Hiernach erfolgt nach der Generierung eines `Stream` ein Mapping mit der Methode `getType` der Klasse `TypableStatement`. Anschließend müssen alle Typargumente in einem zweiten Mapping durch einen `Resolver` aufgelöst werden. Als Rückgabetyp des Methodenaufrufs wird deren Attribut `type` über die Methode `getType` verwendet. Dieser muss vor der heterogenen Übersetzung auch mittels des `Resolver`s aufgelöst werden. Anschließend kann analog zu `LambdaExpression` der Bytecode mithilfe des Interfaces `FunNUtilities` generiert werden. Wie auch bei `LambdaExpression` wurde zuvor für die homogene Übersetzung ausschließlich die Kardinalität von `arglist` verwendet.

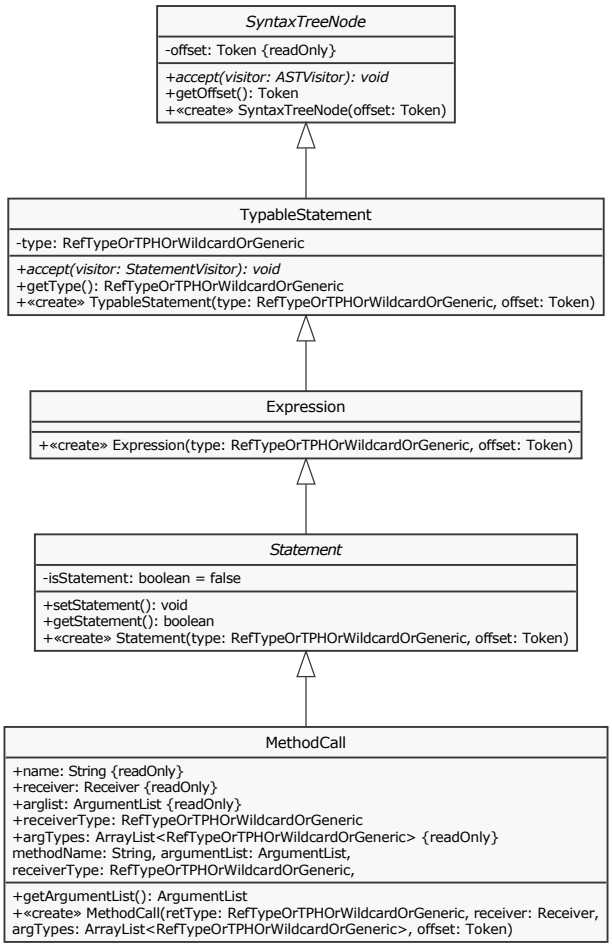


Abbildung 4.4.: Klassendiagramm der Klasse `MethodCall`.

4.4. FunNUtilities

Bei dem Interface `FunNUtilities` handelt es sich um ein neu hinzugefügtes Interface. Dieses wurde im Package `bytecode.funN` hinzugefügt. Das Interface beschreibt dabei die Funktionalitäten die benötigt werden, um den Bytecode für heterogene Funktionstypen zu generieren. Darunter fällt zum einen die Bytecodegenerierung selbst, aber auch die Extraktion der Typargumente und des Rückgabetyps aus einer Liste von Typen. Zudem muss eine Funktionalität bereitgestellt werden, welche spezialisierte Deskriptoren und Signaturen generiert. Die Entscheidung der Bereitstellung dieser Funktionalität im Interface `FunNUtilities` ist in deren Engen Kopplung am Umgang mit TPHs und Generics zu begründen. Außerdem besteht eine enge Kopplung zum spezialisierten Name der heterogenen Funktionstypen. Zukünftig sollte evaluiert werden, ob diese Kopplungen aufgelöst werden können und ein Refactoring angebracht sei. Das Interface `FunNUtilities` wird durch die Klasse `FunNGenerator` implementiert. Ein Klassendiagramm zu `FunNGenerator` ist in Abbildung 4.5 dargestellt.

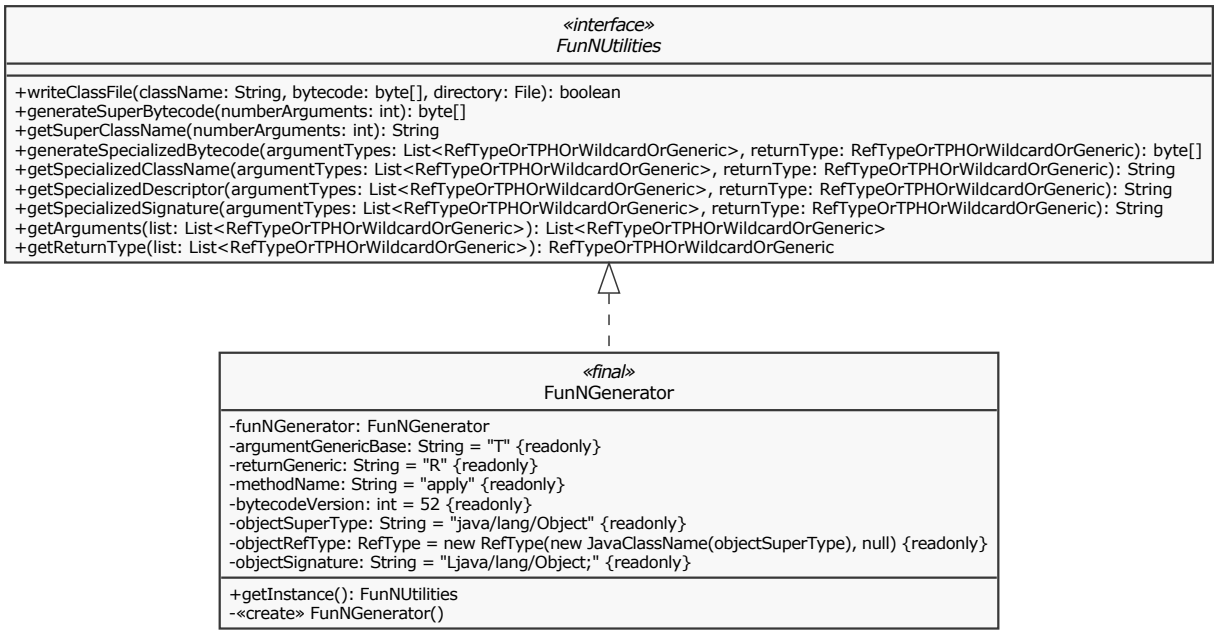


Abbildung 4.5.: Klassendiagramm der Klasse `FunNGenerator`.

`FunNGenerator` implementiert dabei das Interface `FunNUtilities` als eine Art *stateless Singleton Service*. Das private statische Attribut `funNGenerator` instanziiert dabei Thread sicher eine Instanz vom Typ `FunNGenerator`. Durch dessen privaten Konstruktor und der statischen Methode `getInstance` wird die Eigenschaft einer *Singleton* Klasse erreicht. Da `FunNGenerator` keinen mutierbaren internen Zustand besitzt, handelt es sich um eine Thread sichere Klasse. Als Service kann die Klasse betrachtet werden, da diese autark eine Funktionalität bündelt und über die klar definierte Schnittstelle `FunNUtilities` zur Verfügung stellt. Unit Tests wurden zudem zur Qualitätssicherung erstellt. Diese wurden analog zur Package Struktur angelegt. Der Fokus lag hierbei vor allem auf positiven Tests, welche als relevant für die Bytecodegenerierung betrachtet werden. Die Bytecodegenerierung selbst wird mithilfe von direkt durch ASM generierten Bytecode getestet. Dabei wird ein Byte-Array während des Tests mittels ASM generiert und anschließend verglichen. Zudem wurden Ende-zu-Ende-Tests zur bestehenden Test Suite der Bytecodegenerierung hinzugefügt. Fehler welche nicht direkt mit der heterogenen Übersetzung korrelieren, wurden dabei missachtet. Aus diesem Grund mussten Testklassen in Java-TX der Test Suite hinzugefügt werden, für welche aufgrund der Vorgänger:innen korrekter Bytecode generiert werden kann.

5. Reflexion und Ausblick

Das folgende Kapitel beinhaltet eine kritische Reflexion dieser Arbeit. Die Reflexion beschränkt sich dabei auf Kapitel 2 „Generische Parameter“, Kapitel 3 „Umsetzung“ und Kapitel 4 „Implementierung“. Des Weiteren beschränkt sich die Reflexion von Kapitel 2 auf die Themen ab Abschnitt 2.5 „Theoretische Lösung zur Übersetzung echter Funktionstypen“. Diese Beschränkungen werden definiert, da die restlichen Themen keine Eigenleistung sondern ausschließlich Recherche beinhalten.

Im Allgemeinen wurde das Zeil dieser Arbeit erreicht. Das Problem der homogenen Übersetzung echter Funktionstypen konnte gelöst werden. Der Lösungsansatz wurde in Abschnitt 2.5 erörtert. Die Lösung des Problems selbst kann als positives Ergebnis betrachtet werden. Es handelt sich jedoch nicht um eine vollständig erörterte Lösung. Wie der Abschnitt 2.3 zeigt, existieren Sprachen welche die heterogene Übersetzung direkt im Laufzeitsystem adressieren. Für die Reduktion an Komplexität wurde solch ein Ansatz im Rahmen dieser Arbeit nicht betrachtet. Zukünftig wäre es jedoch von Nöten auch solch ein Ansatz zu erörtern und dessen Vor- und Nachteile dar zu legen. Mögliche Punkte dabei wären die Reduktion von Speicherplatz oder Effizienzänderungen in der JVM. Zudem muss erörtert werden, welchen Einfluss eine mögliche Anpassung der JVM auf die Abwärtskompatibilität besitzt.

Darauf aufbauend konnte die theoretische Lösung umgesetzt werden. Positiv ist die geringe Menge Code welche nötig war, um eine entsprechende Übersetzung zu ermöglichen. Ausschließlich die Signatur- und Deskriptor-Generierung und die Bytecodeerstellung mussten angepasst werden. Dies geht einher mit einer klaren Trennung der Funktionalität zu weiteren Komponenten. Dem Gedanke der heterogenen Übersetzung als Implementierungsdetail des Bytecodes konnte somit Rechnung getragen werden. Dennoch besteht zukünftig Verbesserungspotential in der Aufteilung der Komponenten. So sollte allgemein die Bytecodegenerierung in weitere Komponenten untergliedert werden. Ein Vorschlag hierzu wurde in Abbildung 3.2 dargestellt. Außerdem muss evaluiert werden, ob das Interface `FunNUtilities` Funktionalitäten wie `getSpecializedDescriptor` oder `getSpecializedSignature` besitzen sollte. Für die erste prototypische Implementierung ist dies ein valider Ansatz. Möglicherweise sollte jedoch diese Funktionalität ausschließlich in den Klassen `TypeToDescriptor` und `TypeToSignature` realisiert werden. Zudem sollte nach dem Refactoring erneut der Ansatz einer Substitutionskomponente evaluiert werden. Dies konnte im Rahmen der Arbeit nicht durchgeführt werden. Die Substitutionskomponente würde den AST vor der Übergabe an die Bytecodegenerierung abändern. Dabei würden

alle generischen Typen durch deren heterogene Übersetzung substituiert werden. Negativ hieran wäre, dass dies den Gedanken der heterogenen Übersetzung als Implementierungsdetail des Bytecodes verletzt. Begründet wird dies damit, dass die Substitutionskomponente nicht direkt in der Bytecodegenerierung umgesetzt wäre.

Im Rahmen der Arbeit konnte festgestellt werden, dass die aktuelle Test-Suit nicht ausreichend ist. Positive Tests konnten nicht als Implikation für korrekten Bytecode angesehen werden. Manuelle Tests ergaben, dass trotz positiver Tests teilweise inkorrekter Bytecode erstellt wurde. Die davon betroffene Tests waren Tests von vorherigen Programmierenden. Zur Folge hat dies einen ungewissen Status der Qualität der Bytecodegenerierung. Im Rahmen der Arbeit wurde versucht durch Unit-Tests sicher zu stellen, dass dennoch die heterogene Übersetzung in jedem Fall zu keinem neuen inkorrekten Bytecode führt. Anfangs angedachte Tests mussten aus diesen Umständen heraus angepasst werden. Zudem besitzt Java-TX Stand dieser Arbeit noch nicht den vollen Sprachumfang von Java. Aus diesen Gründen müssen zukünftig möglicherweise auch neue Tests für die heterogene Übersetzung hinzugefügt werden.

Eine der wichtigsten Aufgaben in Zukunft ist die Erweiterung der Lösung auf beliebige Typen. So soll zukünftig in Java-TX eine Überladung von Methoden mit den formalen Parametern `List<Integer>` und `List<Double>` möglich sein. Probleme bereiten könnte die nicht vorhandene Mehrfachvererbung in Java. Diese wird zudem planmäßig nicht in Java-TX eingeführt. Somit muss eine unabhängige Lösung gefunden werden. Ein erster Ansatz hiervon wäre ausschließlich die Generierung von heterogenen Übersetzungen für die unterste Hierarchieebene. Dabei wird überprüft, ob die bei der Generierung zu spezialisierenden Klasse bereits eine generische Klasse spezialisiert. Sollte dies der Fall sein, so wird weiterhin eine homogene Übersetzung verwendet. Ansonsten wird eine spezialisierte Subklasse generiert und anstatt dieser verwendet.

6. Zusammenfassung

In diesem Kapitel erfolgt eine kurze Zusammenfassung der Inhalte dieser Arbeit. Zu Beginn der Arbeit wurde Java-TX und die Motivation der Arbeit vorgestellt. Zudem wurde das Ziel dieser Arbeit in Abschnitt 1.3 dargestellt. Darauf Aufbauend wurden im Kapitel 2 die Grundlagen zum Verständnis dieser Arbeit gelegt. So wurde zuerst dargestellt, was generische Parameter sind und anschließend wie diese in Java umgesetzt und verwendet werden. Anschließend zu dieser Vorstellung wurde auch die *Type Erasure* in Java vorgestellt. Bei dieser handelt es sich um eine Abbildung, welche generische Typen in eine homogene Übersetzung abbildet. Die Type Erasure beschreibt dabei das Standard-Verhalten zur Übersetzung generischer Typen in Java. Im Abschnitt 2.3 wurde dargestellt wie die Übersetzung generischer Parameter in Java vergleichbaren Sprachen umgesetzt wird. Betrachtet wurden dabei C# , Pizza und Scala. C# stellte dabei eine Ausnahme dar, da dies die einzige Sprache war, welche nicht auf der JVM ausgeführt wird. Für das Verständnis dieser Arbeit war es zudem von Nöten die neuen Funktionalitäten in Java-TX einzuführen. Dies geschah im Abschnitt 2.4. Anhand dieses Wissens konnte anschließend ein Lösungsvorschlag für eine heterogene Übersetzung echter Funktionstypen abgeleitet werden. Nachzulesen ist dieser im Abschnitt 2.5.

Nach der theoretischen Auseinandersetzung mit der Thematik erfolgte die praktische. Zuerst musste im Kapitel 3 ein Prototyp für die Umsetzung erstellt werden. Dabei wurde mittels ASM probeweise der Bytecode generiert. Dies führte zu einem besseren Verständnis, wie die sogenannte *fluent-API* von ASM hierzu genutzt werden konnte. Für die genaue Umsetzung musste hiernach der aktuelle Stand von Java-TX erörtert und dargestellt werden. Auf Basis dessen konnten mehrere Umsetzungsmöglichkeiten im Abschnitt 3.3 definiert werden. Als umzusetzende Möglichkeit wurde dabei eine ausschließliche Anpassung der Bytecodegenerierung ausgewählt. Vorteil hiervon war die Kapselung der Funktionalität ausschließlich in der Bytecodegenerierung. Somit wurde die heterogene Übersetzung als Implementierungsdetail des Bytecodes aufgefasst. Die endgültige Implementierung wurde anschließend in Kapitel 4 dargestellt. Für die Darstellung der Implementierung war es nötig nicht nur die Änderungen, sondern auch die zu verwendenden Klassen einzuführen. So wurden beispielsweise die AST-Typen in Java-TX oder die Implementierungen des visitor Patterns durch `TypeToDescriptor` oder `TypeToSignature` vorgestellt. Abschließend dazu wurde das neue Interface `FunNUtilities` eingeführt. Dieses bietet Funktionalitäten zur heterogenen Übersetzung echter Funktionstypen an. Die Implementierung dieses Interfaces geschah durch die Klasse `FunNGenerator`.

7. Literaturverzeichnis

Bücher

[Aho08] Aho, A. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium Informatik. Pearson Education Deutschland, 2008. ISBN: 9783827370976. URL: https://books.google.de/books?id=pTKAwL64NkoC.

Online

[Ode+22] Odersky, M. u. a. *Scala Language Specification - Version 2.13*. 2022. URL: https://scala-lang.org/files/archive/spec/2.13/ (besucht am 09. 02. 2022).

[Ora22] Oracle. *The Java Language Environment*. 2022. URL: https://www.oracle.com/java/technologies/introduction-to-java.html (besucht am 25. 01. 2022).

[Bru21] Bruneton, E. *ASM*. 2021. URL: https://asm.ow2.io/index.html (besucht am 17. 02. 2022).

[Mic21a] Microsoft. *A tour of the C# language*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (besucht am 14. 02. 2022).

[Mic21b] Microsoft. *Generics in .NET*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/standard/generics/ (besucht am 14. 02. 2022).

[Ora21a] Oracle. *The Java Language Specification – Java SE 16 Edition*. 2021. URL: https://docs.oracle.com/javase/specs/jls/se16/html/index.html (besucht am 28. 01. 2022).

[Ora21b] Oracle. *The Java Language Specification – Java SE 17 Edition*. 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/html/index.html (besucht am 27. 01. 2022).

[Ora20] Oracle. *The Java Language Specification – Java SE 15 Edition*. 2020. URL: https://docs.oracle.com/javase/specs/jls/se15/html/index.html (besucht am 28. 01. 2022).

[Ora19] Oracle. *Java Developer's Guide*. 2019. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Oracle-JVM-JIT.html#GUID-23D5BA60-A2B3-45F9-93DF-81A3D971CA50 (besucht am 25. 01. 2022).

[Plü18] Plümicke, M. *Das Java-TX Projekt*. 2018. URL: http://www.hb.dhbw-stuttgart.de/~pl/javatx.html (besucht am 19. 01. 2022).

[UW18] Urma, R.-G. und Warburton, R. *Java 10 Local Variable Type Inference*. 2018. URL: https://developer.oracle.com/java/jdk-10-local-variable-type-inference.html (besucht am 03. 02. 2022).

[Ora14] Oracle. *What's New in JDK 8*. 2014. URL: https://www.oracle.com/java/technologies/javase/8-whats-new.html (besucht am 29. 01. 2022).

[Goe13] Goetz, B. *State of the Lambda*. 2013. URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html (besucht am 02. 02. 2022).

[OSV08] Odersky, M.; Spoon, L. und Venners, B. *Programming in Scala, First Edition*. 2008. URL: https://www.artima.com/pins1ed/index.html#TOC (besucht am 09. 02. 2022).

[GH05] Garms, J. und Hanson, T. *Experiences with the New Java 5 Language Features*. 2005. URL: https://www.oracle.com/technical-resources/articles/java/java-5-features3.html (besucht am 23. 01. 2022).

[Jon02] Jones, S. P. *Haskell 98 Language and Libraries*. 2002. URL: https://www.haskell.org/onlinereport/index.html (besucht am 31. 01. 2022).

Konferenz Paper

[PS17] Plümicke, M. und Stadelmeier, A. „Introducing Scala-like Function Types into Java-TX“. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: Association for Computing Machinery, 2017, S. 23–34. ISBN: 9781450353403. DOI: 10.1145/3132190.3132203. URL: https://doi.org/10.1145/3132190.3132203.

[Plü07] Plümicke, M. „Typeless Programming in Java 5.0 with Wildcards“. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. PPPJ '07. Lisboa, Portugal: Association for Computing Machinery, 2007, S. 73–82. ISBN: 9781595936721. DOI: 10.1145/1294325.1294336. URL: https://doi.org/10.1145/1294325.1294336.

[IV02] Igarashi, A. und Viroli, M. „On Variance-Based Subtyping for Parametric Types“. In: *ECOOP 2002 — Object-Oriented Programming*. Hrsg. von Magnusson, B. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 441–469. ISBN: 978-3-540-47993-2.

[KS01] Kennedy, A. und Syme, D. „Design and Implementation of Generics for the .NET Common Language Runtime“. In: *Programming Language Design and Implementation*. ACM Press, Jan. 2001. URL: https://www.microsoft.com/en-us/research/publication/design-and-implementation-of-generics-for-the-net-common-language-runtime/.

[OW97] Odersky, M. und Wadler, P. „Pizza into Java: Translating Theory into Practice“. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, S. 146–159. ISBN: 0897918533. DOI: 10.1145/263699.263715. URL: https://doi.org/10.1145/263699.263715.

Glossar

Kontravarianz

Der Typparameter verhält sich entgegengesetzt zur Vererbungshierarchie.

Kovarianz

Der Typparameter verhält sich gleich zur Vererbungshierarchie.

Legacy Code

Alter Code, welcher meist nicht weiterentwickelt wird.

Anhang

A. Klasse `OLFun` in Java

B. Pizza parametrische Polymorphie

C. Klasse `OLFun` in C#

D. ASM Code des Prototypen

A. Klasse OLFun in Java

Quellcode der Klasse OLFun

```
public class OLFun {

    public Integer m (Fun1$$<Integer,
        Integer> f, Integer x){
        x = f.apply(x+x);
        return x;
    }

    public Double m (Fun1$$<Double, Double>
         f, Double x){
        x = f.apply(x+x);
        return x;
    }
}
```

Listing L.1: Klasse OLFun in Java mit zusätzlichem Parameter x.

Signaturen der Methode m der Klasse OLFun

```
(LFun1$$<Ljava/lang/Double;Ljava/lang/Double;>;Ljava/lang/Double;)
                    Ljava/lang/Double; &
(LFun1$$<Ljava/lang/Integer;Ljava/lang/Integer;>;Ljava/lang/Integer;)
                    Ljava/lang/Integer;
```

Abbildung A.1.: Signaturen der Methode m der Klasse OLFun aus Listing L.1.

Deskriptoren der Methode m der Klasse OLFun

```
(LFun1$$;Ljava/lang/Double;)Ljava/lang/Double &
(LFun1$$;Ljava/lang/Integer;)Ljava/lang/Integer;
```

Abbildung A.2.: Deskriptoren der Methode m der Klasse OLFun aus Listing L.1.

B. Pizza parametrische Polymorphie

Gebundene Polymorphie

```
interface Ord<elem> {
    boolean less(elem o);
}

class Pair<elem implements Ord<elem>> {
    elem x; elem y;
    Pair (elem x, elem y) { this.x = x;
        this.y = y; }
    elem min() {if (x.less(y)) return x;
        else return y; }
}

class OrdInt implements Ord<OrdInt> { int i
    ;
    OrdInt (int i) { this.i = i; }
    int intValue() { return i; }
    boolean less(OrdInt o) { return i < o.
        intValue(); }
}

Pair<OrdInt> p = new Pair(new OrdInt(22),
    new OrdInt(64));
System.out.println(p.min().intValue());
```

Listing L.2: Gebundene Polymorphie in Pizza nach [OW97].

Homogene Übersetzung der gebundene Polymorphie

```
interface Ord {
    boolean less Ord(Object o);
}

class Pair {
    Ord x; Ord y;
    Pair (Ord x, Ord y) { this.x = x; this.
        y = y; }
    Ord min() {if (x.less Ord(y)) return x;
         else return y; }
}

class OrdInt implements Ord {
    int i;
    OrdInt (int i) { this.i = i; }
    int intValue() { return i; }
    boolean less(OrdInt o) { return i < o.
        intValue(); }
```

```
    boolean less_Ord(Object o) { return
        this.less((OrdInt)o); }
}

Pair p = new Pair (new OrdInt(22), new
    OrdInt(64));
System.out.println(((OrdInt)(p.min())).
    intValue());
```

Listing L.3: Homogene Übersetzung der gebundenen Polymorphie nach [OW97].

C. Klasse `OLFun` in C#

```
class OLFun
{
    int m(IFun1<int, int> f) => f.apply(1);
    double m(IFun1<int, int> f) => f.apply
        (2.0);
    string m (IFun1<string, string> f) f.
        apply("Hello World");
    // int m(IFun1<string, string> f) => 1;
         //Error: same parameter types
}
```

Listing L.4: Klasse `OLFun` aus Listing 2.2 in C#.

D. ASM Code des Prototypen

ASM Dumps für die Funktionstypen

```
import org.objectweb.asm.ClassWriter;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

public class Fun1$$Dump implements Opcodes
    {

    public static byte[] dump() throws
        Exception {
        ClassWriter classWriter = new
            ClassWriter(0);
        MethodVisitor methodVisitor;
        classWriter.visit(V1_8, ACC_PUBLIC
            | ACC_ABSTRACT | ACC_INTERFACE,
             "Fun1$$", "<T1:Ljava/lang/
```

```
                Object;R:Ljava/lang/Object;>
                Ljava/lang/Object;", "java/lang
                /Object", null);
            {
                methodVisitor = classWriter.
                    visitMethod(ACC_PUBLIC |
                    ACC_ABSTRACT, "apply", "(
                    Ljava/lang/Object;)Ljava/
                    lang/Object;", "(TT1;)TR;",
                     null);
                methodVisitor.visitEnd();
            }
            classWriter.visitEnd();
            return classWriter.toByteArray();
        }
    }
```

Listing L.5: Basisinterface für Funktionstypen mit einem formalen Parameter.

```
    import org.objectweb.asm.ClassWriter;
    import org.objectweb.asm.Opcodes;

    public class
        Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$Dump
         implements Opcodes {

        public static byte[] dump() throws
            Exception {
            ClassWriter classWriter = new
                ClassWriter(0);
            classWriter.visit(V1_8, ACC_PUBLIC
                | ACC_ABSTRACT | ACC_INTERFACE,
                 "
                Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
                ", "Ljava/lang/Object;LFun1$$<
                Ljava/lang/Integer;Ljava/lang/
                Integer;>;", "java/lang/Object"
                , new String[]{"Fun1$$"});
            classWriter.visitEnd();
            return classWriter.toByteArray();
        }
    }
```

Listing L.6: Spezialisierung von Listing L.5 für `Integer` → `Integer`.

```
    import org.objectweb.asm.ClassWriter;
    import org.objectweb.asm.Opcodes;
```

```
public class
    Fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$Dump
     implements Opcodes {

    public static byte[] dump() throws
        Exception {
        ClassWriter classWriter = new
            ClassWriter(0);
        classWriter.visit(V1_8, ACC_PUBLIC
            | ACC_ABSTRACT | ACC_INTERFACE,
             "
            Fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$
            ", "Ljava/lang/Object;LFun1$$<
            Ljava/lang/Double;Ljava/lang/
            Double;>;", "java/lang/Object",
             new String[]{"Fun1$$"});
        classWriter.visitEnd();
        return classWriter.toByteArray();
    }
}
```

Listing L.7: Spezialisierung von Listing L.5 für `Double` → `Double`.

ASM Code zur Codegenerierung

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class MainGenerate {
    public static void main(String[] args)
        throws Exception{
        var fun1$$ = Fun1$$Dump.dump();
        try(BufferedOutputStream writer =
            new BufferedOutputStream(new
            FileOutputStream("Fun1$$.class"
            ))) {
            writer.write(fun1$$);
            writer.flush();
        }

        var
            fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
             =
            Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$Dump
            .dump();
        try(BufferedOutputStream writer =
            new BufferedOutputStream(new
```

```
                FileOutputStream("
                Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
                .class"))) {
                writer.write(
                    fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
                    );
                writer.flush();
            }

            var
                fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$
                 =
                Fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$Dump
                .dump();
            try(BufferedOutputStream writer =
                new BufferedOutputStream(new
                FileOutputStream("
                Fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$
                .class"))) {
                writer.write(
                    fun1$$Ljava$lang$Double$_$Ljava$lang$Double$_$
                    );
                writer.flush();
            }

            var olFun = OLFunDump.dump();
            try(BufferedOutputStream writer =
                new BufferedOutputStream(new
                FileOutputStream("OLFun.class")
                )) {
                writer.write(olFun);
                writer.flush();
            }
        }
    }
```

Listing L.8: ASM Code zur Generierung der Funktionstypen.