JavaTX
– Extended abstract –

Andreas Stadelmeier
Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
stan@hb.dhbwstuttgart.de
pl@dhbw.de

Abstract. JavaTX steht für Java Type Extended und ist eine Verbindung aus Java und Typinferenz.
Unser Projekt zielt darauf ab die objektorientierte Programmiersprache Java mit einem globalen Typinferenzalgorithmus auszustatten. Dieser erlaubt dem Benutzer sämtliche Typen in seinem Programmcode auszulassen ohne dabei auf die Vorteile einer statisch typisierten objektorientierten Programmiersprache verzichten zu müssen.
Zuletzt führte das JavaTX-Projekt einen Code-Rewrite durch, bei dem der komplette Programmcode von Grund auf neu geschrieben wurde.
Der Vortrag präsentiert das Ergebnis dieses Rewrites und die damit einhergehenden Änderungen und neuen Funktionalitäten des JavaTX-Compilers.

1 JavaTX

Das JavaTX Projekt entwickelt Compiler und Werkzeuge für die Programmiersprache JavaTX. Bei JavaTX handelt es sich um eine Abwandlung der Programmiersprache Java, welche es erlaubt Typen im Quellcode auszulassen.
Das Projekt hat das Ziel die Sprache Java bestmöglich zu imitieren und die meisten seiner Sprachfeatures zu übernehmen. Zusätzlich soll dieser Compiler mit einem Typinferenzalgorithmus gekoppelt werden, welcher im Quellcode ausgelassene Typen selbstständig ermittelt. Dadurch soll es möglich sein Java ähnlich einer dynamischen Programmiersprache ohne die Angaben von Typen zu benutzen, aber immer noch die Vorteile einer statisch typisierten Programmiersprache zu genießen.
Diese Ziele versucht das Projekt mittels eines eigenständigen Compilers zu realisieren. Dieser ist in Java geschrieben und wird weitestgehend über Studienarbeiten vorangetrieben.

1.1 Typinferenz

Es handelt sich bei der von uns verwendeten Typinferenz um globale Typinferenz. Das bedeutet, dass der Algorithmus auch dann zurecht kommt, wenn der Programmierer im gesamten Projekt keinen einzigen Typ angegeben hat.

Bisher wird in der Praxis für Objektorientierte Programmiersprachen lokale Typinferenz eingesetzt. Im Falle von Java ist dies zum Beispiel beim Einsatz des Diamant-Operators wie zum Beispiel in
`List<String> listOfStrings = new ArrayList<>();`
der Fall.
Die Abbildungen 1 und 2 zeigen den Einsatz unseres Typinferenzalgorithmus. Alle in Abbildung 1 grau gefärbten Angaben würden von unserem JavaTX Compiler intern automatisch hinzugefügt und müssten vom Programmierer nicht angegeben werden.

```
interface Fun1{
    A apply(B par1);
}

interface Fun2<A,B,C>{
    A apply(B par1, C par2);
}

class Matrix<A, B>{
    Fun1<Fun1<A, Fun2<A, B, Matrix>>,B>
    op = (m) → (f) → f.apply(m, this);
}
```

Fig. 1. Java **ohne** Typinferenz

```
class Matrix{
    op = (m) -> (f) -> f.apply(m, this);
}
```

Fig. 2. Java **mit** Typinferenz

1.2 Code Rewrite

Die Studienarbeiten, welche den Compiler weiterentwickelten, beschränken sich auf einen Arbeitsaufwand von gerade einmal 150 Stunden. Das führt dazu, dass immer in kleinen Teilprojekten am Compiler gearbeitet wurde, was zu einem chaotischen Zustand des Projekts geführt hat. Ein kompletter Rewrite des Quellcodes war notwendig.
Da der Compiler von Grund auf neu geschrieben nutzen wir dies zum Umbau grundlegender Strukturen. Die dabei entstandenden Änderungen, Verbesserungen und neuen Funktionen sind im folgenden aufgelistet.

Parser mit voller Java 8 Unterstützung: Das Projek stieg auf ANTLR als Parser Generator um. Dieser bringt eine vollständige Grammatik für Java 8 mit sich. Diese musste für JavaTX nur leicht angepasst werden. Der neue Parser unterstützt daher die Sprache Java komplett und akzeptiert zusätzlich dazu auch Java Programme, in denen Typen ausgelassen wurden. Der restliche Teil des JavaTX Compilers unterstützt zwar noch nicht alle Sprachfeatures von Java, hat nun mit diesem Parser allerdings eine starke Basis auf die zukünftige Erweiterungen aufbauen können.

Auftrennung des Compiliervorgangs in einzelne Schritte: *Divide & Conquer* ist eines der entscheidenen Programmierparadigmen vor allem bei Objektorientierten Programmiersprachen wie Java. In unserem Java Projekt sind allerdings über die Zeit einige Aufgaben und Algorithmen zusammengewachsen. Das führte zu einer wirren Struktur und nicht mehr nachvollziehbarem Ablauf des Compiliervorgangs. Der Code Rewrite gab die Chance für eine Umstrukturierung. Das Projekt wurde in verschiedene Aufgaben unterteilt, die alle aufeinander aufbauen und sequentiell abgelaufen werden. Dabei achteten wir auf eine möglichst strikte Trennung der Aufgaben. Das erleichtert das Arbeiten an den einzelnen Aufgaben, da nur dieser eine Teil im Detail verstanden werden muss und nicht das ganze Projekt. Außerdem lassen sich nun Teile des Algorithmus leichter austauschen. Zudem versuchten wir mit der Einführung gewisser Vorgaben, wie dem Immutable AST und der Verwendung des Listener-Patterns, diese Struktur auch für zukünftige Projekte einzuhalten.