

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Informatik

Umsetzung des Java-TX Unifikations-Algorithmus als Webservice

Fabian Holzwarth

31. Juli 2025

Betreuer und Gutachter

Martin Plümicke
Duale Hochschule Baden-Württemberg
DHBW Stuttgart, Campus Horb

Holzwarth, Fabian:

Umsetzung des Java-TX Unifikations-Algorithmus als Webservice

Bachelorarbeit Informatik

Eberhard Karls Universität Tübingen

Matrikelnummer: 6326289

Bearbeitungszeitraum: 01.04.2025 - 01.08.2025

Erklärung

Laut Beschlüssen der Prüfungsausschüsse Bioinformatik, Informatik, Informatik Lehramt, Kognitionswissenschaft, Machine Learning, Medieninformatik und Medizininformatik der Universität Tübingen vom 05.02.2025. Gültig für Abschlussarbeiten (B.Sc./M.Sc./B.Ed./M.Ed.) in den zugehörigen Fächern. Bei Studienarbeiten und Hausarbeiten bitte nach Maßgabe des/der jeweiligen Prüfers/Prüferin.

1. Allgemeine Erklärungen

Hiermit erkläre ich:

- Ich habe die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.
- Ich habe alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
- Die Arbeit war weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens.
- Falls ich ein elektronisches Exemplar und eines oder mehrere gedruckte und gebundene Exemplare eingereicht habe (z.B., weil der/die Prüfer/in(nen) dies wünschen): Das elektronisch eingereichte Exemplar stimmt exakt mit dem bzw. den von mir eingereichten gedruckten und gebundenen Exemplar(en) überein.

2. Erklärung bezüglich Veröffentlichungen

Eine Veröffentlichung ist häufig ein Qualitätsmerkmal (z.B. bei Veröffentlichung in Fachzeitschrift, Konferenz, Preprint, etc.). Sie muss aber korrekt angegeben werden. Bitte kreuzen Sie die für Ihre Arbeit zutreffende Variante an:

- ☒ Die Arbeit wurde bisher weder vollständig noch in Teilen veröffentlicht.
- ☐ Die Arbeit wurde in Teilen oder vollständig schon veröffentlicht. Hierfür findet sich im Anhang eine vollständige Tabelle mit bibliographischen Angaben.

3. Nutzung von Methoden der künstlichen Intelligenz (KI, z.B. chatGPT, DeepL, etc.)

Die Nutzung von KI kann sinnvoll sein. Sie muss aber korrekt angegeben werden und kann die Schwerpunkte bei der Bewertung der Arbeit beeinflussen. Bitte kreuzen Sie alle für Ihre Arbeit zutreffenden Varianten an und beachten Sie, dass die Varianten 3.4 - 3.6 eine vorherige Absprache mit dem/der Betreuer/in voraussetzen:

- ☐ 3.1. Keine Nutzung: Ich habe zur Erstellung meiner Arbeit keine KI benutzt.
- ☒ 3.2. Korrektur Rechtschreibung & Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

- ☐ 3.3. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.
- ☐ 3.4. Übersetzung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Übersetzung von mir in einer anderen Sprache geschriebenen Texte genutzt. Jede derartige Übersetzung ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller übersetzten Textstellen und der verwendeten Programme mit Versionsnummer.
- ☐ 3.5. Code-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Code in der Softwareentwicklung genutzt. Der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.
- ☐ 3.6. Text-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Text in meiner Arbeit genutzt. Jede derartige Verwendung von KI ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

Falls ich in irgendeiner Form KI genutzt haben (siehe oben), dann erkläre ich:

Mir ist bewusst, dass ich die Verantwortung trage, falls es durch die Verwendung von KI zu fehlerhaften Inhalten, zu Verstößen gegen das Datenschutzrecht, Urheberrecht oder zu wissenschaftlichem Fehlverhalten (z.B. Plagiaten) kommt.

4. Abschluss und Unterschrift(en)

Mir ist bekannt, dass ein Verstoß gegen diese Erklärung prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass die Prüfungsleistung mit „nicht ausreichend“ bzw. die Studienleistung mit „nicht bestanden“ bewertet wird und bei mehrfachem oder schwerwiegendem Täuschungsversuch eine Exmatrikulation erfolgen bzw. ein Verfahren zur Entziehung eines eventuell verliehenen akademischen Titels eingeleitet werden kann.

<u>Fabian, Holzwarth</u>	<u>Reutlingen, 31.07.2025</u>	<u>F. Holzwarth</u>
Vorname, Nachname Student/in	Ort, Datum	Unterschrift

Die Punkte 3.4 - 3.6 erfordern eine Zustimmung des/r Betreuer/in. Sollten Sie einen dieser Punkte angekreuzt haben, dann sollte der/die Betreuer/in bitte hier unterschreiben:

Ich habe der oben genannten Nutzung von KI zur Erstellung der Arbeit zugestimmt.

_____	_____	_____
Vorname, Nachname Betreuer/in	Ort, Datum	Unterschrift

Zusammenfassung

Java-TX ist eine Programmiersprache mit zugehörigem Compiler auf Basis der Sprache Java, die an der DHBW entwickelt wird. Zu den wichtigsten Funktionen zählt die Implementierung echter Funktionstypen und einer globalen Typ-Inferenz. Die explizite Definition von Datentypen soll ohne Verlust der strikten Typsicherheit inferiert und angewandt werden.

Ein zentraler Teil der Typ-Inferenz ist der Unifikations-Algorithmus, welcher basierend auf den Strukturen des generischen Java aus einer Menge von Typ-Bedingungen (Type-Constraints) alle möglichen Lösungen berechnet. Im Rahmen einiger früherer Arbeiten wurde dieser Algorithmus bereits erweitert, in [Ste15] neu implementiert und in [Leh23] die parallele Ausführung überarbeitet. Allerdings existieren noch immer bei Fällen mit vielen möglichen und stark verschachtelter Typisierungen ein hoher zeitlicher Aufwand zur Ausführung des Unifikations-Algorithmus. Das erschwert die Nutzung des Compilers und stellt ein Hindernis für eine effiziente Nutzung von Java-TX dar.

Diese Arbeit stellt einige Analysen und Verbesserungen des Algorithmus sowie die Implementierung eines Webservices zur externen Berechnung mit entsprechender Hardware vor. Aufwendige Berechnungen sollen somit anstelle des lokalen Geräts einen hochgradig parallelen Rechner nutzen, wodurch eine deutliche Verbesserung der Performance erhofft wird.

Inhaltsverzeichnis

Abbildungsverzeichnis	v
Tabellenverzeichnis	vii
Abkürzungsverzeichnis	viii
1 Einleitung	1
1.1 Motivation	1
1.2 Thema der Arbeit	1
1.3 Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Programmiersprachen	3
2.1.1 Typsystem des generischen Java	4
2.1.2 Typinferenz in Java-TX	7
2.2 Die Server-Client-Architektur	9
2.2.1 Webservices	9
2.2.2 Websockets	10
2.2.3 Das Datenformat JSON	10
2.3 Laufzeitoptimierung	12
2.3.1 Nebenläufigkeit	12
2.3.2 Speicheroptimierung	12
3 Java-TX Unifikations-Algorithmus	13

3.1	Typ-Bedingungen und Typ-Platzhalter	14
3.2	Typ-Unifikation	15
3.3	Der Unifikations-Algorithmus	15
3.4	Implementierung im Java-TX-Compiler	17
4	Laufzeit-Analyse und Optimierungen	19
4.1	Laufzeitanalyse	19
4.1.1	Testfälle	21
4.2	Benchmark-Tests	21
4.3	Analyse und Optimierungen	22
4.3.1	Thread-Synchronisation	23
4.3.2	Platzhalter-Generierung	23
4.3.3	Varianz-Trennung	25
4.3.4	Task-Switching in ForkJoinPool	27
4.3.5	Parallele Null-Varianz Berechnung	29
4.3.6	Logging-Optimierung	30
4.3.7	HashMap Presizing	30
4.4	Optimierungsergebnisse	31
5	Umsetzung des Webservice	32
5.1	Server-Konzeption	32
5.2	Vorbereitungen	34
5.2.1	Kapselung des Unifikations-Algorithmus	35
5.2.2	Kapselung der Platzhalter Generierung	35
5.2.3	Kapselung von Parametern	36
5.2.4	Implementierung eines Loggers	37
5.3	Implementierung des WebSocket-Servers	39
5.3.1	Server-Endpunkt Implementierung	40
5.3.2	Daten-Serialisierung zu JSON	41
5.3.3	Implementierung eines Packet-Systems	45

5.4	Ergebnis der Server-Implementierung	45
5.4.1	Zeitmessung mit Webservice	47
6	Tests - Abläufe und Ergebnisse	48
6.1	Unit-Tests	48
6.2	Integrations-Tests	48
7	Zusammenfassung	49
7.1	Ergebnis	50
7.2	Ausblick	50
	Quellenverzeichnis	51
A	Performance-Test Beispiele	53
A.1	Matrix.jav	54
A.2	GridSearch.jav	55
A.3	Convolution.jav	56
B	Server-Implementierung	57
B.1	Interface für serialisierbare Klassen	57
B.2	Serialisierung einer Java-Klasse	58
B.3	Implementierung einer IPacket-Klasse	59
C	Test-Implementierung	60
C.1	Paket-Tests zur (De-)Serialisierung	60

Abbildungsverzeichnis

2.1	Type Erasure	5
2.2	Das Kontravarianz-Problem	7
2.3	Java-TX Code-Beispiel	8
2.4	Java-TX Ergebnis-Beispiel	8
2.5	Rekursive JSON-Struktur nach [Int17]	11
3.1	Klasse: TypeUnifyTask (vereinfacht)	17
3.2	Klasse: TypeUnify2Task (vereinfacht)	18
3.3	TypeUnifyTask Rekursions-Ablauf	18
4.1	Heatmap des Master-Branches	20
4.2	Überflüssige Verwendung von synchronized Schlüsselwörtern	23
4.3	Bisherige Platzhalter-Generierung	23
4.4	Aktualisierte Platzhalter Generierung	24
4.5	Varianz-Basierte Fallunterscheidung	25
4.6	Separierte Varianz-Klassen	26
4.7	Ursprüngliches Fork-Join-Modell	28
4.8	Geändertes Future-Modell	28
4.9	Lazy Evaluation von Logging-Ausgaben	30
4.10	Heatmap mit allen Optimierungen für das Matrix-Beispiel	31
5.1	Vereinfachte Version der Unify-Context Klasse	37
5.2	Logger Nutzung und Ausgaben	38
5.3	Farb-Schema des Loggers	38

5.4	Interface: ISerialNode	42
5.5	Beispiel: JSON Darstellung eines Pair-Objekts	43
5.6	Beispiel: JSON Darstellung der Klasse Pair	44
5.7	Packet-System Interfaces	45
5.8	Webservice Ablauf	46
A.1	Beispiel-Datei 1: Matrix.jav	54
A.2	Beispiel-Datei 2: GridSearch.jav	55
A.3	Beispiel-Datei 3: Convolution.jav	56
B.1	Interface: ISerializableData	57
B.2	Beispiel: Serialisierung in Pair.java	58
B.3	Packet-Beispiel: MessagePacket	59
C.1	Test-Beispiel: Unify-Pair Serialisierung	60

Tabellenverzeichnis

4.1	Benchmark-Tests: Master-Branch	22
4.2	Zeitmessung: Umstellung auf Future-Design	29
4.3	Zeitmessung: Logging-Optimierung	30
4.4	Zeitmessung: HashMap-Presizing	31
5.1	Zeitmessung: Master-Branch auf dem Test-Server	47
5.2	Zeitmessung: Optimierter Code auf dem Test-Server	47
5.3	Zeitmessung: Lokale Ausführung mit Webservice	47

Abkürzungsverzeichnis

Java-TX	Java Type-Extended
TPH	Type-Placeholder (Typ-Platzhalter)
JVM	Java-Virtual-Machine
API	Application-Programming-Interface
HTTP	HyperText-Transfer-Protocol
JSON	JavaScript-Object-Notation
TCP	Transmission-Control-Protocol
UDP	User-Diagram-Protocol
URL	Uniform-Resource-Locator

Kapitel 1

Einleitung

1.1 Motivation

Der Java-TX-Compiler [DHBW25] ist das Kernstück der Programmiersprache Java-TX und für die Übersetzung des Quellcodes zu Binärcode für die JVM (Java-Virtual-Machine) zuständig. Entsprechend relevant ist die Performance des Programms, um in angemessener Zeit ein korrektes Ergebnis zu erhalten. Der Hauptteil der Rechenzeit ist aktuell auf den komplexen Vorgang der globalen Typinferenz zurückzuführen. Je nach Eingabe und Hardware sind Laufzeiten im Bereich von Minuten möglich, was bei Prozessen wie Kompilation, Testing oder Autovervollständigung für sehr hohe Latenz sorgt.

1.2 Thema der Arbeit

Das Ziel dieser Arbeit ist primär die Implementierung eines Webservices, um komplexe Berechnungen auf einem stark parallelisierten Server ausführen zu können. Zusätzlich zu den erhofften Performance-Gewinnen besteht als sekundäres Ziel die allgemeine Verbesserung der Performance des Unifikation-Algorithmus.

Im ersten Schritt wird daher der bestehende Stand des Algorithmus mit den Optimierungen aus der vorhergehenden Arbeit [Leh23] geprüft und gefundene Schwachstellen angepasst. Dies dient als Vorbereitung auf den zweiten Schritt, in dem der Algorithmus vom Rest des Programms isoliert und als Webservice auf einem massiv-parallelen-Rechner ausgeführt werden soll. Durch die hohe Anzahl an verfügbaren Threads werden deutliche Verbesserungen der Berechnungszeit erhofft.

1.3 Aufbau der Arbeit

Zu Beginn werden in Kapitel 1 einige Begriffe und Konzepte ausgeführt, welche für diese Arbeit als Grundlagen dienen. Dabei wird der Fokus auf der technischen Seite des Algorithmus und den Server-Konzepten liegen, da der theoretische Hintergrund der Unifikation in diesem Kontext nur bedingt relevant ist.

Anschließend folgt in Kapitel 4 die Analyse der Code-Basis aus [Leh23] und der zu Beginn dieser Arbeit aktuellsten Version des Compilers. Gefundene Schwachstellen sollen verbessert und die allgemeine Performance erhöht werden.

Darauf aufbauend wird in Abschnitt 5.3.1 ein Konzept für eine Serverseitige Anwendung erarbeitet und nachfolgend implementiert. Diese Anwendung soll vom Compiler des Clients zur Ausführung einzelner Tasks verwendet werden können. In erster Linie umfassen diese Tasks die Ausführung des Unifikations-Algorithmus, aufgrund dessen hoher Komplexität.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige Begriffe und Konzepte erläutert, die als Grundlage für die Durchführung dieser Arbeit dienen.

Zuerst werden in Abschnitt 2.1 die Grundlagen des Java-TX Projekts in Vorbereitung auf die Analyse in Kapitel 4 erläutert.

Im Anschluss folgen in Abschnitt 2.2 Erklärungen zu den Bestandteilen eines Webservice in Bezug auf die Umsetzung in Kapitel 5.

2.1 Programmiersprachen

Programmiersprachen bilden im Allgemeinen ein Interface für die Erzeugung von ausführbarem Programmcode. Sie steigern die Schreib- und Lesbarkeit von Programmen durch Menschen mit der Einführung von sprachlichen Bezeichnern, ausführlicherer Syntax und Semantik. Die Eigenschaften der Daten werden dafür in der Sprache als Datentypen abstrahiert, um anschließend vom Compiler oder Interpreter in die passenden Instruktionen übersetzt zu werden. Die Implementierung von Datentypen kann sich von Sprache zu Sprache unterscheiden und lässt sich hauptsächlich anhand von drei Eigenschaften beschreiben:

- Explizite oder implizite Typisierung
Diese Eigenschaft beschreibt die Art der Herleitung von Typ-Definitionen im Programmcode. Datentypen müssen entweder bei expliziter Typisierung vom Programmierer angegeben werden, oder können bei impliziter Typisierung anhand der bekannten Definitionen und Zuweisungen vom Compiler ermittelt werden. Oft werden auch Mischformen unterstützt, bei denen manche Datentypen anhand anderer Definitionen erkannt werden können.

- **Nominale oder strukturelle Typisierung**
Im Falle von nominaler Typisierung wird ein Datentyp und dessen Beziehung zu anderen Datentypen durch einen eindeutigen Namen gekennzeichnet. Das Gegenteil bildet eine strukturelle Typisierung, die einen Datentyp durch dessen Aufbau definiert und somit verschiedene aber gleichwertige Bezeichner erlaubt. Java verwendet durchweg nominale Typisierung.
- **Statische oder dynamische Typisierung**
Durch diese Eigenschaft wird der Zeitpunkt der Typisierung ausgedrückt. Die statische Typisierung wird bereits zur Kompilierung ausgewertet. Somit können viele Typ-Fehler bereits vor der Ausführung entdeckt oder die zusätzlichen Informationen zur Optimierung des generierten Codes genutzt werden. Dynamische Typisierung führt die Typ-Prüfung hingegen erst zur Laufzeit aus. Code kann dadurch mit weniger Aufwand verfasst oder angepasst werden und kann auch einfacher auf eine größere Menge an verschiedenen Datentypen reagieren. Es entsteht jedoch ein erhöhtes Risiko von Typ-Fehlern und einem zusätzlichen Rechenaufwand zur Laufzeit.

2.1.1 Typsystem des generischen Java

Die Programmiersprache Java ist explizit, nominal und statisch typisiert. Alle Datentypen müssen daher explizit vom Programmierer vor dem Kompilieren festgelegt werden. Allerdings bieten aktuelle Java Versionen für lokale Variablen mittels des “var”-Schlüsselwort sowie dem Diamant-Operator $T<>()$ auch eine simple lokale Typinferenz. Das Typ-System baut in der aktuellen Version auf Datentypen auf, die sich grob in drei Arten unterteilen lassen:

- **Primitive Datentypen** Diese Datentypen repräsentieren direkte Daten, wie diese später von der JVM gehandhabt werden und unterscheiden sich nur in der Speichergröße (z.B. byte mit 8 Bits und int mit 32 Bits) und Verwendung (z.B. short als signed 16 Bits und char als unsigned 16 Bits).
- **Konkrete Klassen** Bei konkreten Klassen handelt es sich um Konstrukte der Programmiersprache, welche mehrere Daten und Code kapseln und zusammenfassen. Durch diese ist es möglich, die Verwendung von Code oder den Zugriff auf Daten besser zu kontrollieren und zusammengehörende Daten zu kombinieren. Für die Kompatibilität mit primitiven Typen existiert in Java das Boxing-Prinzip. Damit werden bei Bedarf primitive Typen in ihre Box-Klassen oder zurückkonvertiert, was sowohl implizit als auch explizit passieren kann.

```
1  class MyClass {
2      int value;
3      int method() { return 1; }
4  }
5
6  MyClass myclass = new MyClass();
```

- Generische Klassen Als Erweiterung und Obermenge der konkreten Klassen wurden generische Klassen eingeführt. Diese erhalten eine Liste an Klassen-Typen als Typ-Parameter übergeben, anhand derer Eigenschaften und Methoden der Klasse für den jeweiligen Typ-Parameter ausgelegt werden. Die Zuweisung von Typ-Parametern existiert in Java ausschließlich beim Kompilieren, da generische Parameter zur Laufzeit durch den Vorgang der Type-Erasure entfernt werden. Dies dient vor allem der Rückwärts-Kompatibilität, da generische Parameter erst in späteren Versionen implementiert wurden.

```
1  class MyGenericClass<A,B> {
2      A value;
3      B method(B param) { ... }
4  }
5
6  MyGenericClass<String, Integer> myclass
7      = new MyGenericClass<String, Integer>();
8
9  // Wird zur Laufzeit zu:
10 // MyGenericClass myclass = new MyGenericClass();
```

Abbildung 2.1: Type Erasure

Typ-Parameter können auch durch Wildcards Abhängigkeiten erzeugen und die erlaubten Klassen eingrenzen. Dabei kann mittels “?” eine beliebige, unbekannte Klasse ausgedrückt werden.

Java-Klassen bilden eine eindeutige und baumartige Hierarchie, wobei jede Klasse von einer Elternklasse erbt und jede Elternklasse eine beliebige Anzahl Kindklassen haben kann. Ausnahme dafür ist nur die abstrakte Mehrfachvererbung durch Interfaces und die Wurzelklasse `java.lang.Object`, welche

durch Vererbung eine Elternklasse jeder anderen Klasse ist und selbst keine Elternklasse besitzt. Falls nicht anders angegeben, ist diese die implizite Elternklasse. Klassen werden durch abstrakte Klassen und Interfaces erweitert, welche Klassen-Definitionen ohne oder mit unvollständiger Implementierung zulassen aber nicht instanziiert werden können.

In Java 8 wurde der hauptsächlich objektorientierte Ansatz durch die Implementierung von Lambda-Ausdrücken in Kombination mit Funktionalen Interfaces erweitert. Damit ist funktionale Programmierung auch in Java ohne Einsatz von expliziten Wrapper-Klassen möglich und deutlich weniger aufwendig.

Varianz

Das Verhältnis zwischen zwei Datentypen kann durch das Konzept von Varianz ausgedrückt werden. Je nach Varianz können unterschiedliche Annahmen über die Beziehung der Daten zueinander gemacht werden. So können Datentypen vollständig unabhängig voneinander sein und werden als invariant bezeichnet. Besteht eine Beziehung in Richtung der Vererbung, ist ein Datentyp kovariant zu einem anderen. Im gegenteiligen Fall ist der Datentyp kontravariant.

Existieren zum Beispiel zwei Klassen A und B, wobei B von A erbt ($B < A$). Im Kontext einer überschriebenen Methode kann der Rückgabewert nur Kovariant definiert werden. Hat die ursprüngliche Methode also Typ A zurück gegeben, kann die überschreibende Methode nun B zurückgeben.

Unter der Anwendung von Wildcards können generische Datentypen darüber hinaus auch in ihrem Parameter Kovarianz und auch Kontravarianz erzeugen. Ansonsten sind zwei generische Klassen mit verschiedenen Typ-Parametern in Java immer invariant, unabhängig der tatsächlichen Beziehung der Typ-Parameter. Daraus folgt, dass zum Beispiel `List<Object>` keine Elternklasse von `List<String>` ist, obwohl `Object` eine Elternklasse von `String` ist. Zuweisungen wie die Folgende sind daher unzulässig:

```
1 class Vector<A> {}  
2 Vector<Object> objs = new Vector<String>();
```

Das lässt sich auf das Kontravarianz-Problem zurückführen. Existiert zwischen zwei Typen eine Kontravarianz-Beziehung, gehen auf einer Seite Typ-Details verloren und es besteht das Risiko von Laufzeitfehlern. In [Plü04] wird das Problem mit folgendem Beispiel veranschaulicht:

```
1 class Super { ...}
2 class Sub extends Super { ...}
3 class Vector<A> {
4     void add (A elem) { ...}
5 }
6
7 class Main {
8     public static void main(String[] args) {
9         Vector<Super> v;
10        v = new Vector<Sub> ();
11        // Folgendes würde nun einen Laufzeitfehler erzeugen,
12        // kompiliert aber erfolgreich
13        v.add(new Super());
14    }
15 }
```

Abbildung 2.2: Das Kontravarianz-Problem

Dieses Problem wird in Java durch die generelle Invarianz von Typ-Parametern umgangen. Kovarianz und Kontravarianz lassen sich seit der Einführung von generischen Klassen in Java 5 durch die Nutzung des Wildcard Operators mit “? extends T” (Kovarianz) und “? super T” (Kontravarianz) ausdrücken.

2.1.2 Typinferenz in Java-TX

In Java-TX wird das Typsystem von Java um eine globale und implizite Typinferenz erweitert. Im Gegensatz zu lokaler Typinferenz, bei der lediglich die unmittelbar bekannten Typen verwendet werden, werden bei der globalen Variante auch Typen über mehrere Definitionen hinweg inferiert. Ein Datentyp hängt daher nicht mehr nur von den Definitionen der verwendeten Variablen, Klassen und Methoden ab, sondern kann auch über die Verwendung innerhalb dieser bestimmt werden.

Das macht es in Java-TX möglich, die Typ-Angabe für Variablen, Methoden-Parameter und Methoden-Rückgabewerte auszulassen und automatisch anhand der importierten Klassen, Zuweisungen und Nutzung zu inferieren. Zur Einschränkung der möglichen Klassen werden jedoch nur explizit importierte Klassen bei der Inferenz berücksichtigt. So wird die Gesamtmenge der Lösungen begrenzt und Mehrdeutigkeit verringert. Eine Klasse kann also wie folgt definiert werden:

```
1 import java.util.ArrayList;
2 import java.lang.Integer;
3
4 class AddToList {
5     apply(a, b) {
6         a.add(b);
7         return b;
8     }
9
10    applyPlusOne(a, b) {
11        return 1 + apply(a, b);
12    }
13 }
```

Abbildung 2.3: Java-TX Code-Beispiel

Der Compiler erzeugt daraufhin die folgende Klassen-Definition (dekompiliert aus Byte-Code)

```
1 import java.util.ArrayList;
2
3 class AddToList {
4     public AddToList() {}
5
6     Integer apply(ArrayList<Number> var1, Integer var2) {
7         var1.add(var2);
8         return var2;
9     }
10
11    Integer applyPlusOne(ArrayList<Number> var1, Integer var2) {
12        return 1 + this.apply(var1, var2);
13    }
14 }
```

Abbildung 2.4: Java-TX Ergebnis-Beispiel

Dabei ist anzumerken, dass auch mehrere Lösungen existieren können und der in Kapitel 3 beschriebene Algorithmus potentiell alle Lösungen finden kann. Daher können zwei Aufrufe des Compilers für dieselbe Datei zwei unterschiedliche, aber jeweils gültige Belegungen finden. Welche Belegung gewählt wird, ist aufgrund von Laufzeitoptimierungen und der Nutzung von nicht deterministischen Klassen wie `java.util.HashSet` allerdings nicht vorauszusagen.

2.2 Die Server-Client-Architektur

Während herkömmliche Prozesse auf einem einzelnen Computer ausgeführt werden, kann dieses Prinzip durch die Verbindung mehrerer Computer erweitert werden. Dabei wird eine Verbindung zwischen mehreren Computern aufgebaut, über die Daten ausgetauscht werden können. Heutzutage sind entfernte Verbindungen üblich und funktionieren als Grundlage des Internets und anderer dezentraler Anwendungen.

Um die Kommunikation solcher Verbindungen zu erleichtern, kommt in den meisten Fällen die Server-Client-Architektur zum Einsatz. Diese unterscheidet sich von Peer-to-Peer Verbindungen durch das Zuweisen spezieller Rollen zu den einzelnen Computern. So existieren genaue Erwartungen an die andere Seite und es kommt zu keiner Datenkollision oder falschen Annahmen gegenüber dem Verbindungspartner. Der Computer mit der Rolle des Servers stellt dabei in der Regel die passivere Seite beim Aufbau von Verbindungen dar. Der Server wartet durchgehend auf den Eingang von Verbindungsanfragen des Clients und beantwortet diese gegebenenfalls. Auf der anderen Seite steht der Client-Computer, welcher aktiv einen Verbindungsaufbau zum Server startet und von diesem eine bestimmte Antwort oder Aufgabe fordert.

2.2.1 Webservices

Als Teil der Server-Client-Architektur ist die Umsetzung von Webservices möglich. Ein Webservice stellt eine auf Software basierende Schnittstelle für Clients zur Verfügung, die sowohl für den isolierten Zugriff auf externe Systeme, als auch für das Bereitstellen von vordefinierten Funktionen verwendet werden können. In dieser Arbeit wird ein Webservice umgesetzt, der Funktionen zur Ausführung der Typ-Inferenz für den Client mittels eigener Rechenleistung bereitstellt.

2.2.2 Websockets

Ein typisches Protokoll für Datenübertragung im Internet ist das HyperText-Transfer-Protokoll (kurz HTTP) über das Transport-Control-Protocol (kurz TCP). HTTP-Anfragen sind in der Regel kurzlebig, zustandslos und folgen dem Request-Response-Prinzip. Dabei erfolgt auf jede Anfrage des Clients genau eine Antwort des Servers. Für die Übertragung mehrerer unabhängiger Daten wurde das Protokoll in HTTP/2 durch Streams erweitert, die das Aufteilen einer Anfrage in mehrere Pakete erlauben. Eine vollkommen bidirektionale Kommunikation ist damit aber auch nur eingeschränkt möglich.

Im Gegensatz dazu bleibt die TCP-Verbindung bei Anfragen über das WebSocket Protokoll [AM11] nach dem Verbindungsaufbau dauerhaft geöffnet, bis diese erst zu einem späteren Zeitpunkt von einer der beiden Seiten oder einem Verbindungsabbruch geschlossen wird. Durch die bestehende TCP-Verbindung ist eine beidseitige und beliebig ausführliche Kommunikation zwischen Client und Server möglich. Anstelle einer einzelnen Antwort kann der Server auch mit einem Datenpaket reagieren, das vom Client wiederum beantwortet werden kann. So ist ein komplexerer Datenaustausch zwischen den Computern möglich, ohne dafür immer neue TCP Verbindungen aufbauen zu müssen. Im Vergleich mit Zwischenlösungen wie regelmäßigen HTTP-Anfragen bei HTTP-Polling zeigen Websockets eine deutlichere Verringerung der benötigten Datenmenge und somit eine bessere Performance.

In dieser Arbeit wird das Prinzip von Websockets für die Kommunikation zwischen Client und Server verwendet, um dem Server die Möglichkeit von mehreren unmittelbaren Antworten zur Laufzeit des Client-Prozesses zu geben. Gleichzeitig legt das auch den Grundstein für mögliche Erweiterungen, bei denen der Client mehrere Anfragen zur Berechnung an den Server schickt.

2.2.3 Das Datenformat JSON

Das Datenformat JSON, kurz für “JavaScript Object Notation”, ist eine textbasierte Darstellungsart für strukturierte Daten. Dem Namen entsprechend geht JSON aus den Objekt-Literalen in JavaScript hervor und wurde dann in der ECMAScript Spezifikation 404 [Int17] definiert. Das Format teilt sich in die folgenden Arten von Werten, die rekursiv miteinander kombiniert werden können:

- Objekte, welche als Schlüssel-Wert Paare dargestellt werden. Schlüssel sind immer Text-Strings in Anführungszeichen, während die Werte eine beliebige Mischung aus den erlaubten Typen beinhalten können.
- Listen, welche eine geordnete Menge von Werten beinhalten. Diese Werte können ebenfalls eine beliebige Mischung aus den erlaubten Typen sein.
- Zeichenketten (Strings), welche durch doppelte Anführungszeichen gekennzeichnet sind.
- Zahlen, die entweder als ganze Zahl $\mathbb{Z}$ oder als rationale Zahl $\mathbb{Q}$ mit Dezimal-Punkt angegeben werden können. Führende Nullen sind nicht erlaubt, allerdings ein führendes Minus-Zeichen oder die Exponenten-Schreibweise.
- Eines der Literale "true", "false" oder "null".

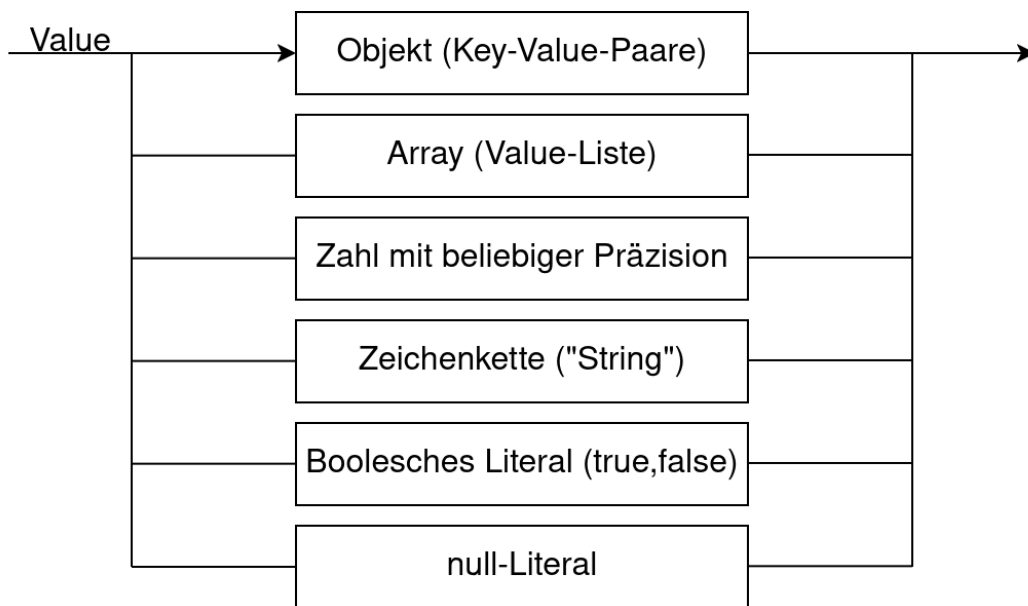


Abbildung 2.5: Rekursive JSON-Struktur nach [Int17]

2.3 Laufzeitoptimierung

Die Dauer der Berechnung eines Programms kann auf unterschiedliche Arten verbessert werden. So kann das zu berechnende Problem eingeschränkt werden und es ist weniger Aufwand zur Lösung notwendig. Alternativ kann auch das Vorgehen bei der Berechnung optimiert werden, sodass das selbe Problem einen geringeren Aufwand zur Berechnung benötigt. Diese Arbeit fokussiert sich auf die Optimierung der Berechnung anstelle der Einschränkung des Problems. Darauf wird in Kapitel 4 jedoch genauer eingegangen.

2.3.1 Nebenläufigkeit

Ein Ansatz zur Steigerung der Performance ist die Nutzung von Nebenläufigkeit. Moderne Computer haben in der Regel mehrere CPU-Kerne eingebaut, die gleichzeitig unterschiedliche Berechnungen durchführen und so die Arbeitslast aufspalten können. Über die Abstraktionsebene von Threads kann ein Prozess einen Teilprozess auf einem eigenen Thread starten und nach dessen Beendigung wieder mit sich zusammen führen. Bei der Nutzung von Nebenläufigkeit (auch Multi-Threading genannt) können allerdings auch problematische Situationen entstehen. So erfordert das Nutzen mehrerer Threads einen organisatorischen Aufwand und erlaubt auch das Auftreten von Race-Conditions zwischen zwei Berechnungen. Andere Teile des Programms müssen sich zudem daran anpassen, was zum Beispiel auf die Garbage-Collection zutrifft.

Ein verwandtes Konzept von Multi-Threading in Java ist die Nutzung von Futures. Diese stellen eine spezielle Menge an Klassen dar, die Folgeberechnungen in Lambda-Ausdrücken speichert und so einen Teil des Programms im Continuation-Passing-Style sequentiell ausdrücken kann. Die Ausführung der Teil-Ausdrücke kann dabei auch auf mehrere Threads aufgeteilt werden, da der ursprüngliche Code nicht ein Ergebnis warten muss.

2.3.2 Speicheroptimierung

Ein anderer wichtiger Aspekt bei Laufzeitoptimierungen ist die Optimierung von Speicher und Caches. Große Mengen an einzelnen Daten steigern den Aufwand bei der Garbage-Collection und erschweren zudem die Nutzung von Caches auf Hardware-Ebene. Der Nutzen von neuen Objekten muss besonders in häufig genutzten Funktionen gegen den entstehenden Zusatzaufwand abgewogen werden.

Kapitel 3

Java-TX Unifikations-Algorithmus

Als inhaltliche Basis für die Umsetzungen dieser Arbeit wird nun kurz auf den Unifikations-Algorithmus in JAVA-TX eingegangen. Hier wird das Konzept nur grob erläutert, da für diese Arbeit lediglich der Ablauf der Unifikation relevant ist.

Wie in [Plü09] beschrieben, wird als Grundlage für die Unifikation im Typ-System von Java die durch die Klassen-Hierarchie definierte Typ-Ordnung verwendet. Darauf aufbauend ergibt sich eine Closure (Abschluss-Menge) mit allen Relationen zwischen je zwei Elementen der Typ-Ordnung. Durch generische Klassen ergibt sich das Problem einer potenziell unendlich großen Menge an Relationen, welche in der Praxis nicht dargestellt werden können. Als Lösung definieren wir eine endliche Hülle (“Finite Closure”), welche eine ausreichende Teilmenge aller Relationen beinhaltet und durch Reflexivität und Transitivität auch Rückschlüsse über die nicht dargestellten Relationen zulässt.

In der Umsetzung aus [Ste15] wird entsprechend dieser Grundlage die transitive Hülle des definierten Klassen-Baums als Teilmenge genutzt.

3.1 Typ-Bedingungen und Typ-Platzhalter

In Vorbereitung auf die Inferenz der Typen werden die verfügbaren Informationen aus dem Quell-Code extrahiert. Neben den Expliziten Typ-Definitionen werden Typ-Platzhalter (“Type-Placeholders” oder kurz TPHs) für alle unbekannten Datentypen vergeben. Anhand der Nutzung und der gegenseitigen Abhängigkeit werden dann Typ-Bedingungen (“Type-Constraints”) erstellt.

Eine Typ-Bedingung stellt ein Paar von Typ-Termen dar, welches mit einem Typ-Operator verknüpft ist. Die verwendeten Typ-Terme können dabei explizite Definitionen, Typ-Platzhalter oder eine Kombination daraus sein. Jede Bedingung besteht damit aus einem Paar von Typ-Termen (θ_1, θ_2) und einem Operator $\otimes \in \{=, <, <?\}$, wodurch eine Relation der Form $\theta_1 \otimes \theta_2$ entsteht. Die Operatoren werden wie folgt definiert [Plü23]:

- $=$: Die Typ-Terme θ_1 und θ_2 müssen gleich sein. Für eine Lösung muss also eine Zuweisung gefunden werden, durch die beide Terme identisch werden.
- $<$: Hierbei wird zwischen θ_1 und θ_2 eine Subtyp Relation gefordert, wobei θ_1 ein Subtyp von θ_2 oder gleich θ_2 sein muss.
- $<?$: Mit einem generischen umschließenden Typ C muss die Bedingung $C<\theta_1> < C<\theta_2>$ gelten. Durch die generelle Invarianz von generischen Typen spielt dieser Fall hauptsächlich bei der Nutzung von Wildcards eine Rolle.

3.2 Typ-Unifikation

Das Ziel der Typ-Unifikation ist das Finden eines Unifikators (“Unifiers”), unter dessen Anwendung alle gegebenen Bedingungen erfüllt sind und er somit eine Lösung der Typ-Belegung für alle Typ-Platzhalter angibt. Ein Unifikator entspricht einer Menge von Constraints der Form $a \doteq \theta$. Dabei müssen für zwei Constraints die Terme a_1 und a_2 paarweise verschiedenen TPHs entsprechen und in allen Constraints θ ein Typ-Term ohne Bezug auf a sein. Ein Unifikator ist also dann eine gültige und vollständige Lösung, wenn nach der Anwendung weder Widersprüche aufgetreten sind, noch TPHs in den Constraints verbleiben. Der Unifikator mit der geringsten Menge an Zuweisungen wird auch der allgemeinste Unifikator (“most general unifier” oder kurz MGU) genannt.

Im Allgemeinen kann ein Unifikations-Problem “nullary” sein, also keine Lösung haben. Bei genau einem MGU, der bis auf Umbenennung der Variablen eindeutig ist, wird das Problem als “unitary” bezeichnet. Existieren mehrere MGUs, wird dies als “finitary” bezeichnet. Und sollte eine unendliche Menge an MGUs existieren, so wird das Problem als “infinitary”) bezeichnet [Sie89]. Im unserem Kontext der Datentypen in Java ist das Unifikations-Problem durch die Einschränkung der Finite Closure daher allgemein finitary, wobei auch nur ein oder kein MGU für eine bestimmte Instanz des Problems existieren kann.

3.3 Der Unifikations-Algorithmus

In [Plü09] wird ein Algorithmus definiert, welcher die Definition aus [Plü04] um Wildcards erweitert. Dieser Algorithmus baut auf den zuvor beschriebenen Konzepten der Unifikation auf und bietet eine vollständige Lösung für das Unifikations-Problem in Java. Um eine Basis für die Umsetzung dieser Arbeit zu bieten, wird der Ablauf in 7 Schritten kurz zusammengefasst. Die genaue Definition der einzelnen Schritte ist in [Plü09] nachzulesen. In Abschnitt 3.4 wird die Implementierung dieses Algorithmus weiter dargestellt.

Es sei Eq eine Menge von Constraints der Form $\theta_1 \otimes \theta_2$.

1. Anwendung von Reduktions-Regeln zur Vereinfachung der Gleichungen, solange diese angewendet werden können.
2. Es wird die Menge Eq'_1 gebildet, welche alle Constraints mit zwei Typ-Platzhaltern und somit der Form $TPH \otimes TPH$, beinhalten.
3. Es wird die Menge $Eq'_2 = Eq \setminus Eq'_1$ gebildet, in der bei jeder Constraint maximal ein Seite ein Typ-Platzhalter ist.
4. Es wird die Menge Eq'_{set} , welche mittels des kartesischen Produkt alle gültigen Kombinationen aus Eq'_1 und Teilmengen von Eq'_2 erzeugt.
5. Für jede Menge in Eq'_{set} : Anwendung von Unifikatoren (Constraints der Form $a \doteq \theta$, wobei a nicht in θ vor kommt) auf die anderen Elemente der Menge.
6. a) Beginne für jede Menge $Eq' \in Eq'_{set}$, bei der zuvor eine Substitution durchgeführt wurde, neu mit Schritt 1.
 b) Vereinige alle Ergebnisse von 6a) mit allen Mengen, bei denen keine Substitution durchgeführt wurde zu Eq''_{set} .
7. Erhalte einen Unifikator für jede Menge von Eq''_{set} , die in gelöster Form vorliegt, wobei jede der Constraints $a \doteq \theta$ zu einer Unifikations-Regel $a \rightarrow \theta$ wird.

Insbesondere das kartesische Produkt in Schritt 4 kann die Anzahl der zu prüfenden Fälle exponentiell steigern. Durch den rekursiven Aufruf in Schritt 6b wird das weiter gefördert, da jede Kombination wieder einen eigenen Pfad beginnt, in dem wieder neue Kombinationen gebildet werden können.

Als Ergebnis sind sind mehrere gültige Unifikatoren möglich, wenn diese in unterschiedlichen Fällen des kartesischen Produkts zu Lösungen geführt haben. Auch identische Lösungen können auftreten, wenn mehrere Pfade zum selben Ergebnis führen. Sollte ein Widerspruch auftreten, wird der aktuelle Pfad verworfen und mit den verbleibenden Pfaden fortgefahren. In [Plü23] wird zudem ein Backtracking-Ansatz vorgestellt, durch den ein Widerspruch auch alle folgenden, vom selben Widerspruch betroffenen, Pfade verwirft. Sollten alle Mengen verworfen werden, kann keine Unifikation gefunden werden und es kommt zu einem Typ-Fehler.

3.4 Implementierung im Java-TX-Compiler

Im Compiler [DHBW25] wird die Unifikation im package “de.dhbwstuttgart.typeinference.unify” dargestellt. Der Algorithmus selbst wird hauptsächlich durch die Klassen `TypeUnifyTask` und `TypeUnifyTask2` implementiert, wobei diverse Helfer-Klassen zum Einsatz kommen. Im Folgenden werden die zwei Haupt-Klassen in Bezug auf den Algorithmus aus Abschnitt 3.3 vorgestellt.

```

1 public class TypeUnifyTask extends RecursiveTask<Set<Set<UnifyPair>>>
2 {
3     Set<Set<UnifyPair>> compute() {
4         return unify(...);
5     }
6
7     protected Set<Set<UnifyPair>> unify(...) {
8         // Schritte 1-4 und Bildung des kartesischen Produkts
9         return computeCartesianRecursive(...);
10    }
11
12    protected Set<Set<UnifyPair>> computeCartesianRecursive(...) {
13        while(...) { // Für jeden Pfad wird ein neuer Fork erstellt
14            TypeUnify2Task fork = new TypeUnify2Task(...);
15            forks.add(fork); fork.fork();
16        }
17        for(TypeUnify2Task fork : forks) {
18            var fork_result = fork.join();
19        }
20        return results;
21    }
22
23    protected Set<Set<UnifyPair>> unify2(...) {
24        // Schritt 5
25        // Schritt 6a: Rekursion
26        unify(...);
27        // Schritt 6b: Zusammenführung der Ergebnisse
28        return ...;
29    }
30 }

```

Abbildung 3.1: Klasse: `TypeUnifyTask` (vereinfacht)

```

1 public class TypeUnify2Task extends TypeUnifyTask {
2     protected Set<Set<UnifyPair>> compute() {
3         return unify2(...);
4     }
5 }

```

Abbildung 3.2: Klasse: TypeUnify2Task (vereinfacht)

In der Klassen-Definition ist die Elternklasse `RecursiveTask<T>` zu sehen, über welche das Fork-Join-Framework implementiert wurde. Bei einem Fork der entsprechenden Klasse wird im neuen Thread mittels der Methode `T compute()` eingestiegen. Das Ergebnis, der generische Parameter, ist hierbei eine Menge von Unifikatoren.

Als Einstiegspunkt dient die `compute` Methode der `TypeUnifyTask` Klasse. Im Anschluss wird nach Anwendung der Reduktions-Regeln und dem Aufteilen der Constraints die Elemente des kartesischen Produkts gebildet und über neue Objekte der Klasse `TypeUnify2Task` weiter verarbeitet. Als Kontext für die Parallelisierung wird ein Objekt vom Typ `ForkJoinPool` verwendet, das die Zuweisung von Threads koordiniert und durch Work-Stealing das bearbeiten von Aufgaben aus anderen Threads erlaubt.

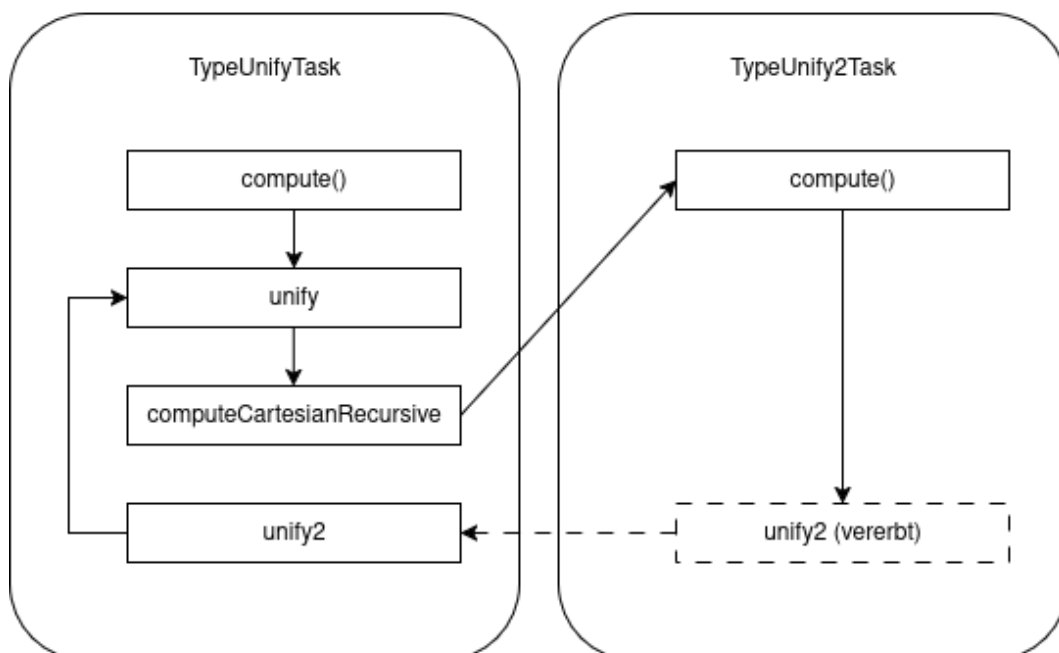


Abbildung 3.3: TypeUnifyTask Rekursions-Ablauf

Kapitel 4

Laufzeit-Analyse und Optimierungen

Ursprünglich war vorgesehen, auf Basis der Arbeit [Leh23] zu arbeiten und zu optimieren. Aufgrund einiger Probleme mit dem darin enthaltenen Algorithmus und fehlschlagenden Tests (siehe Abschnitt 4.2) wurde entschieden, stattdessen auf den ursprünglichen Code der aktuell eingesetzten Version zurückzugreifen. Im Folgenden wird daher [Leh23] als Bezeichner für den dortigen Code verwendet, während für die tatsächlich verwendete Code-Basis die aus dem Git-Repository entnommene Bezeichnung “master-Branch” verwendet wird und dessen Stand bei Commit `1391206dfe` bezeichnet.

Die in [Leh23] eingeführten sinnvollen Erweiterungen wurden im Rahmen dieser Arbeit dennoch aufgegriffen und auf dem master-Branch neu implementiert.

4.1 Laufzeitanalyse

Für die Analyse des Java-Programms wurde VisualVM [Ora25] für Profiling und Überwachung der JVM verwendet. Die Erweiterung “Threads Inspector” wurde zudem installiert und genutzt, um die aktuellen Stacktraces auf Threads einzusehen. Damit war nicht nur ein Einblick in die Thread-Nutzung möglich, sondern es konnten auch mittels Sampling die zeitintensiven Bereiche des Algorithmus beobachtet und identifiziert werden. Für weitere Messungen und die Erstellung von Flamegraphs und Heatmaps wurde zudem das Programm `async-profiler-4.0` [And25] hinzugezogen. Eine Heatmap auf Basis des master-Branchs kann beispielsweise wie in Abb. 4.1 aussehen. Zur besseren Übersicht wurde außerdem eine vergrößerte Teil-Ansicht hinzugefügt:

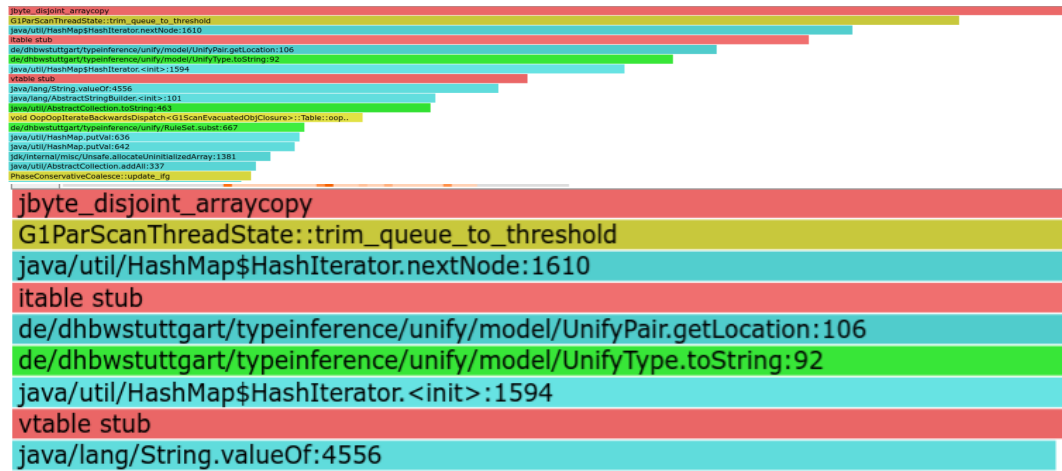


Abbildung 4.1: Heatmap des Master-Branches

Jede Zeile steht hierbei für einen Prozess oder einen Methodenaufruf und die Breite des Balkens für den Anteil an der Gesamtdauer der Messung. Ein breiter Balken deutet also auf einen höheren Anteil an der Gesamtberechnung hin. Die Farben zeigen die Zugehörigkeit des Prozesses. Relevant sind hier der Garbage-Collector in Gelb, native JVM Methoden in Rot und normale Methoden in Blau und Grün. Hier sind bereits einige Probleme deutlich zu erkennen, die in den nachfolgenden Abschnitten behandelt werden. Diese beinhalten unter anderem:

- Kopieren von Arrays, üblicherweise durch Vergrößern von ArrayList oder HashMap Objekten
- Garbage-Collection Thread-Scanning durch viele kurzlebige Objekte
- Aufruf der HashMap-Iterator Methode “nextNode”
- itable und vtable durch Zugriffe auf Methoden aus Interfaces und Überladungen
- Aufrufe der Methode String.valueOf und den spezifischen toString Methoden

4.1.1 Testfälle

Für die Messung der Performance-Änderungen wurden mehrere Code-Beispiele ausgewählt. Da die Komplexität der Unifikation besonders mit steigender Verschachtlung von generischen Typen, wurde darauf der Fokus gelegt. Eine lange Ausführungsdauer erlaubt bessere Einsichten in die problematischen Bereiche und Änderungen in der Laufzeit, als es bei sehr kurzer Ausführungsdauer der Fall ist. Dateien mit flacher Typ-Verschachtlung können üblicherweise in unter einer Sekunde ausgewertet werden und eignen sich daher nicht für die Messung von Unterschieden.

Nach diesen Kriterien wurden drei Beispiele für die Messungen ausgewählt. Dabei handelt es sich um die Datei “Matrix.jav” welche in Anhang A.1 dargestellt wird und bereits als Test-Datei vorhanden war. Zusätzlich wurden zwei neue Beispiele hinzugefügt, welche eine ähnliche Unifikations-Komplexität zur Matrix-Klasse aufweisen und daher ebenfalls eine höhere Verarbeitungsdauer besitzen. Bei diesen handelt es sich um die Dateien “GridSearch.jav” (siehe Anhang A.2) und “Convolution.jav” (siehe Anhang A.3).

Weitere Tests wurden unter anderem mit den Projekt-Internen JUnit-Tests durchgeführt, bei denen jedoch analoge Zeiten ermittelt wurden. Alle Zeit-Angaben werden sich daher lediglich auf die oben stehenden Beispiele konzentrieren. Die Laufzeit-Tests wurden mit den folgenden Befehlen durchgeführt:

```
mvn clean compile package
time java -jar <Compiler-JAR-Pfad> <JAV-Pfad> -d ./targetTest
```

Das Programm `time` wurde zur Ermittlung der realen Ausführungsdauer des Prozesses verwendet. Die Kompilierung des Java-Codes wird dabei nicht zur Zeitmessung addiert, sodass die reine Ausführungszeit gemessen wird. Alle folgenden Zeit-Angaben werden im Format “Minuten:Sekunden” gemacht und bei Bedarf auf die nächste Sekunde gerundet.

4.2 Benchmark-Tests

Als Ausgangspunkt wurden die Tests auf dem master-Branch durchgeführt. Die Ergebnisse variieren zwischen verschiedener Hardware stark. Alle im folgenden gemessenen Zeiten wurden einheitlich auf einem Gerät festgestellt, um Abweichungen durch Hardware auszuschließen. Auf den alternativen Geräten wurden Zeiten mit ähnlichem Verhältnis zueinander gemessen, wobei die CPU der stärkste Einflussfaktor zu sein scheint.

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	01:55 min	01:52 min	01:54 min	01:54 min
GridSearch	02:10 min	02:02 min	02:13 min	02:08 min
Convolution	01:11 min	00:58 min	01:07 min	01:05 min

Tabelle 4.1: Benchmark-Tests: Master-Branch

Dieselben Tests wurden auf dem [Leh23]-Branch ausgeführt. Während einige Beispiele erfolgreich kompiliert wurden, kam es bei den größeren Test-Fällen zu Fehlern. Die beschriebene Klasse Matrix endet mit dem Fehler “Unresolved Constraints”, während GridSearch und Convolution mit einer “NullPointerException” enden. Bei der Ausführung der im Projekt enthaltenen Unit-Tests (mittels dem Befehl “mvn clean compile test”) wurde der Test nach 60 Minuten manuell abgebrochen, da bis zu diesem Zeitpunkt kein Ergebnis berechnet wurde.

4.3 Analyse und Optimierungen

Ursprünglich war vorgesehen, direkt auf dem Code von [Leh23] aufzubauen. Wie aber im vorherigen Abschnitt festgestellt, ist diese Umsetzung nur zu Teilen funktionsfähig. Während einige konkrete Beispiele auf ausreichender Hardware schneller als auf dem master-Branch berechnet werden, laufen andere für unbestimmte Zeit oder werden aufgrund Laufzeit-Fehlern gar nicht kompiliert. Über VisualVM konnten zwei Probleme bei der Ausführungszeit festgestellt werden:

- Beinahe alle Stichproben der Stacktraces zur Programmlaufzeit zeigen Arbeit an HashSets an. Durch die Änderungen scheint in manchen Fällen daher die Anzahl der Ergebnisse und deren Zusammenführungen drastisch anzusteigen.
- Auf Geräten mit einer kleineren Zahl an CPU-Kernen (in diesem Fall: 4) kam es zu durchgehend blockierten Threads, welche durch die häufigen Aufrufe der join Methode ausgelöst wurden.

Zusätzlich wird das Entfernen der Varianz-Optimierung für einige Anpassungen beschrieben, was ebenfalls einen negativen Einfluss auf die Ergebnisse und die Laufzeit hat. Da ein ausführliches Debugging aller Änderungen aufwendiger wäre als eine teilweise Neu-Implementierung, wurde entschieden, die Arbeit stattdessen auf Basis des master-Branches fortzuführen. Sinnvolle Anpassungen wurden nach der Vorlage von [Leh23] übernommen und in einigen Fällen erweitert.

4.3.1 Thread-Synchronisation

Im gesamten Typ-Unifikations-Prozess sind Code-Blöcke mit dem Schlüsselwort “synchronized” versehen, obwohl dieses nicht benötigt wird. In den meisten Fällen entsteht dadurch nur ein Overhead und Thread-Blocking, die für die Ausführung nicht nötig wären. Ein häufiges Beispiel ist die Überprüfung des UnifyTaskModels “usedTasks” zur Überprüfung auf Abbrüche. Das ganze Objekt für die Dauer der Operation zu reservieren hält keinerlei Vorteile, da dieser Wert höchstens einmal geschrieben wird. Entsprechend der Vorlage aus [Leh23] wurden diese Synchronisierungen weitestgehend entfernt. Der resultierende Performance-Gewinn ist minimal, jedoch wird die Leserlichkeit des Codes verbessert und potentielle Bottlenecks entfernt.

```
1 synchronized (usedTasks) {
2     if (this.myIsCancelled()) // Abfrage eines lokalen booleans
3         return new HashSet<>();
4     else
5         return res;
6 }
```

Abbildung 4.2: Überflüssige Verwendung von synchronized Schlüsselwörtern

4.3.2 Platzhalter-Generierung

Für die Generierung von neuen Typ-Platzhaltern (TPH) wurde bislang im master-Branch folgender Code verwendet:

```
1 static final ArrayList<String> EXISTING_PLACEHOLDERS
2     = new ArrayList<String>();
3
4 public synchronized static PlaceholderType freshPlaceholder() {
5     String name = nextName + (char) (rnd.nextInt(22) + 97);
6     while(EXISTING_PLACEHOLDERS.contains(name)) {
7         name += (char) (rnd.nextInt(22) + 97);
8     }
9     return new PlaceholderType(name, true);
10 }
```

Abbildung 4.3: Bisherige Platzhalter-Generierung

Es wird so lange ein zufälliger Großbuchstabe an den Namen angehängt, bis ein unbekannter Platzhalter gefunden wurde. Wie in [Leh23] bereits festgestellt und angepasst wurde, benötigt das konstante Prüfen der EXISTING_PLACEHOLDERS Liste bei vielen Platzhaltern viel Zeit. Da die Methode zudem mit dem synchronized Schlüsselwort versehen ist, blockiert das auch für alle anderen Threads die Platzhalter-Generierung. Hier wurde die ArrayList durch ein Thread-sicheres Set ausgetauscht, sodass die Methode auch parallel erlaubt werden kann. Das wurde in die neue Implementierung übernommen.

Die Erzeugung einer neuen Typvariable ist dennoch weiterhin mit mehrfacher String-Konkatenation, Aufrufen des Pseudo-Zufallsgenerators (PRNG) und Aufrufen der “contains” Methode im Set verbunden. Um weiterhin ein deterministisches Ergebnis zu erhalten, wurde der Zufalls-Generator mit einem festen Seed erstellt. Dieser Vorgang wurde durch eine einfachere Methode ersetzt:

```
1 private static final Set<String> existingPlaceholders
2     = ConcurrentHashMap.newKeySet();
3 private static final AtomicInteger placeholderCount
4     = new AtomicInteger();
5
6 public static String generateFreshPlaceholderName() {
7     String name;
8     do {
9         int pc = placeholderCount.incrementAndGet();
10        name = getUppercaseTokenFromInt(pc);
11    }
12    while (existingPlaceholders.contains(name));
13    return name;
14 }
```

Abbildung 4.4: Aktualisierte Platzhalter Generierung

Diese Änderung reduziert den synchronisierten Bereich auf die Inkrementierung und Abfrage des AtomicInteger. Der restliche Rechenaufwand wird zudem auf die String-Erzeugung beschränkt, welche die neue Zahl als Wert zur Basis 26 mit den Zeichen A-Z ($A = 0, Z = 25$) interpretiert. Sofern Platzhalter ausschließlich über diese Methode erzeugt werden, sinkt die Ausführungsdauer auf eine beinahe konstante Zeit. Die umschließende While-Schleife wurde für einige wenige Sonderfälle beibehalten, um Doppelungen zu vermeiden.

In Abschnitt 5.2.2 wird später auf weitere Änderungen bezüglich der Platzhalter-Generierung eingegangen, die auch diese Sonderfälle weiter einschränken. Die Verbesserung der Performance in den Tests ist eher gering und nicht eindeutig sichtbar. Jedoch ist bei steigender Größe des Projekts und somit einer steigenden Zahl von TPHs mit größeren Gewinnen zu rechnen.

4.3.3 Varianz-Trennung

Innerhalb des Unifikations-Algorithmus gibt es mehrere If-Else Statements, welche sich oft über sehr große Code-Abschnitte ziehen. Das Kriterium ist dabei immer die Varianz. Diese kommt aber nur in wenigen diskreten Werten vor (0,1,-1,2) und eignet sich daher sehr gut für die Abstraktion über Vererbung. Bislang wurde in der Berechnung des kartesischen Produktes folgender (vereinfachter) Code verwendet:

```

1      int variance = ...;
2      while (nextSetasList.size() > 0) {
3          ...
4          if (variance == 1) { ... }
5          else if (variance == -1) { ... }
6          else if (variance == 2) { ... }
7          else if (variance == 0) { ... }
8          ...
9          if (parallel && variance == 1) { ... }
10         else if (parallel && variance == -1) { ... }
11         else if (parallel && variance == 2) { ... }
12         else { ... }
13         ...
14         if (variance == 1) { ... }
15         else if (variance == -1) { ... }
16         else if (variance == 2) { ... }
17         else if (variance == 0) { ... }
18         ...
19         if (variance == 1) { ... }
20         else if (variance == -1) { ... }
21         else if (variance == 2) { ... }
22         else if (variance == 0) { ... }
23     }

```

Abbildung 4.5: Varianz-Basierte Fallunterscheidung

Daher wurden die Branches der If-Else Statements in eigene Klassen verschoben, welche je für einen Varianz-Wert die gemeinsame Elternklasse `VarianceCase` erweitern.

```

1 public abstract class VarianceCase {
2
3     public static VarianceCase createFromVariance(int variance, ...) {
4         return switch (variance) {
5             case 0    -> new UnknownVarianceCase(...);
6             case 1    -> new ContravarianceCase(...);
7             case -1   -> new CovarianceCase(...);
8             case 2    -> new OderConstraintCase(...);
9             default   ->
10                throw new RuntimeException("Invalid variance: " + variance);
11        };
12    }
13
14    // If-Else-Block dargestellt durch eine Methode
15    public abstract void selectNextConstraint(...);
16    ...
17 }
18
19 public class UnknownVarianceCase extends VarianceCase {...}
20 public class ContravarianceCase extends VarianceCase {...}
21 public class CovarianceCase extends VarianceCase {...}
22 public class OderConstraintCase extends VarianceCase {...}

```

Abbildung 4.6: Separierte Varianz-Klassen

Jeder If-Else Block wird anschließend durch eine Methode der neuen Klassen ersetzt und die Fallunterscheidung durch die Abstraktion ersetzt. Eine Verbesserung der Performance ist hierdurch nicht zu erwarten, jedoch wird der Code durch die Abstrahierung, die starke Verkürzung und Präzisierung übersichtlicher und lesbarer.

Diese Implementierung wird aktuell nur für Logik innerhalb des kartesischen Produkts genutzt, bietet aber potentiell auch die Grundlage für das vollständige Ersetzen des primitiven Integer Wertes für die Varianz.

4.3.4 Task-Switching in ForkJoinPool

Die Parallelisierung wurde in der bisherigen Umsetzung über das Fork-Join-Framework gehandhabt. In [Leh23] wurde darauf aufbauend ein eigens erstellter ForkJoinPool implementiert, um mehr Kontrolle über das Verhalten der Parallelisierung zu erhalten. Damit ist in den meisten Fällen mit einem sehr effizienten Multi-Threading zu rechnen. Ein ForkJoinPool implementiert außerdem den Mechanismus des Work-Stealing, durch den die erzeugten Worker-Threads die Tasks anderer Threads stehlen und bearbeiten können, sollten sie selbst keine offenen Aufgaben mehr haben.

Die folgenden Geschwindigkeitstests zeigten jedoch im Vergleich eine Verbesserung der Performance-Tests um nahezu 200% durch die Nutzung von Futures für die Parallelisierung. Der ForkJoinPool wurde als ExecutorService für die Futures wiederverwendet. Beobachtungen aus der Heatmap und der Thread-Nutzung lassen zwei Ursachen vermuten:

- ForkJoinPools sind auf Divide-and-Conquer Strategien ausgelegt, bei denen beide Teil-Bäume der Berechnung in etwa einen ähnlichen Aufwand haben. Bei der aktuellen Nutzung entstehen aber oft stark unausgeglichene Bäume. Der Arbeitsaufwand wird daher ungleich aufgeteilt und Work-Stealing kann nicht effizient eingesetzt werden.
- Durch die Work-Stealing und Scheduling Mechaniken entsteht im Fork-JoinPool ein gewisser Overhead für jeden Task-Wechsel. Die Tasks der Typ-Unifikation haben in den meisten Fällen einen sehr kleinen Arbeitsaufwand, ehe neue Sub-Tasks erstellt werden. Somit steigt der Threading-Overhead stärker an als eine weitere Berechnung dauern würde. Durch die Nutzung von Futures ist kein Work-Stealing mehr möglich. Stattdessen kann bei Beendigung eines Sub-Tasks die Callback-Funktion ohne Overhead auf dem aktuellen Thread ausgeführt werden.

Die Einführung von Futures bringt aber auch Nachteile mit sich. Neben der steigenden Komplexität des Quellcodes wächst auch die Stack-Trace mit jeder zusätzlichen Callback-Funktion. In den Tests konnte dabei jedoch noch kein problematisches Ausmaß festgestellt werden.

Nachfolgend wird die Änderung von ForkJoin zu Futures in einem vereinfachten Beispiel dargestellt.

```

1 List<TypeUnify2Task> forks = new ArrayList<>();
2 Set<Set<UnifyType>> result = new HashSet<>();
3
4 while (!nextSetAsListRest.isEmpty()) {
5     TypeUnify2Task task = new TypeUnify2Task(...);
6     task.fork();
7     forks.add(task);
8 }
9
10 for (TypeUnify2Task task : forks) {
11     var taskResult = task.join();
12     result = combine(result, taskResult)
13 }
14
15 return result;

```

Abbildung 4.7: Ursprüngliches Fork-Join-Modell

```

1 CompletableFuture<...> resultFuture
2     = CompletableFuture.completedFuture(new HashSet<>());
3
4 while (!nextSetAsListRest.isEmpty()) {
5     TypeUnify2Task task = new TypeUnify2Task(...);
6     CompletableFuture taskFuture = CompletableFuture.supplyAsync(
7         task::compute, unifyContext.executor()
8     );
9
10    resultFuture = resultFuture.thenCombine(taskFuture,
11        (result, taskResult) -> {
12            return handle(combine, taskResult);
13        });
14 }
15
16 return resultFuture;

```

Abbildung 4.8: Geändertes Future-Modell

Nach der Änderung von Fork-Join zum Future-basierten Modell zeigt sich in den Tests bereits eine deutliche Verbesserung für alle Beispiele mit geschachtelten Typen.

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	01:14 min	01:17 min	00:58 min	01:09 min
GridSearch	00:57 min	00:55 min	00:55 min	00:55 min
Convolution	00:08 min	00:08 min	00:08 min	00:08 min

Tabelle 4.2: Zeitmessung: Umstellung auf Future-Design

4.3.5 Parallele Null-Varianz Berechnung

Im Falle einer Varianz von 0, also unbekannter oder gemischter Varianz, sieht der bisherige Algorithmus eine serielle Berechnung vor. Zur Optimierung werden die Ergebnisse dann verwendet, um zukünftige Fälle bereits vor der Ausführung auszuschließen. Sollte es sich um eine Oder-Constraint handeln, wird sogar bereits nach dem ersten gültigen Ergebnis abgebrochen.

Besonders im Beispiel der Matrix-Klasse kommt diese Varianz hauptsächlich vor, wodurch die Parallelität nicht ausgenutzt werden kann. Daher wurde versucht, auch bei einer Varianz von 0 die Optimierung durch Berechnung aller Fälle zu ersetzen.

Die Änderung zu einer Parallelen Umsetzung erfordert allerdings Anpassungen an der Zusammenführung, da diese nun nicht mehr wie zuvor in mehreren Etappen abgearbeitet werden können, sondern zukünftige Iterationen vorweg und bei Bedarf abbrechen müssen. Auch eine Behandlung der einzelnen Resultate muss die korrekte Reihenfolge beibehalten. Diese Bedingungen müssen erfüllt sein, damit weiterhin das korrekte Ergebnis geliefert werden kann.

Dadurch ergab sich allerdings wieder eine Verschlechterung der Performance bis einige Tests manuell abgebrochen werden mussten, weshalb diese Änderung nicht übernommen wurde. Zudem kam es zu großen Schwankungen in der Ausführungsdauer bei erfolgreichen Tests. Mögliche Ursachen umfassen die geringe Größe der Berechnung bei Oder-Constraints, welche mehr Thread-Overhead erzeugt als durch die Vorausberechnung eingespart wird oder die deutlich wachsende Anzahl an Thread-Tasks und deren Management. Dieser Umstand tritt besonders auf Geräten mit schwächerer Hardware auf, wo Thread-Wechsel und Garbage-Collection einen größeren Einfluss auf die Laufzeit des Programms haben.

4.3.6 Logging-Optimierung

In Abschnitt 5.2.4 wird später die Implementierung eines einheitlichen Logging-Systems beschrieben. Aus dieser Änderung konnte auch ein Performance-Boost generiert werden, indem aufwendige Logging Aufrufe in Lambda-Funktionen verpackt werden. Diese können dann bei Bedarf ausgeführt und ein String generiert werden, oder andernfalls ohne Auswertung ignoriert werden. Durch den Einsatz dieser Lazy Evaluation Methode wird die aufwendige toString Methode von großen Objekten nicht mehr aufgerufen, wenn diese nicht auch ausgegeben wird. Auch hier war ein deutlicher Gewinn in der Performance sichtbar.

```
1 logger.debug(() -> "Complex data: " + complex.toString());
```

Abbildung 4.9: Lazy Evaluation von Logging-Ausgaben

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	00:42 min	00:39 min	00:44 min	00:42 min
GridSearch	00:33 min	00:32 min	00:33 min	00:33 min
Convolution	00:07 min	00:07 min	00:07 min	00:07 min

Tabelle 4.3: Zeitmessung: Logging-Optimierung

4.3.7 HashMap Presizing

An vielen Stellen im Code werden HashMaps ohne Parameter erzeugt und anschließend mit Elementen befüllt. Dabei werden die HashMaps standardmäßig mit Platz für 16 Elemente erstellt. Überschreitet die Anzahl der Elemente einen Threshold (typischerweise 75%), wird mehr Speicher alloziert und alle existierenden Elemente in den neuen Speicher übertragen. Dafür muss der Hash aller Objekte neu generiert werden, um diese korrekt in die existierenden Buckets einsortieren zu können. Da viele Objekte in der Typ-Unifikation eigene hash-Methoden haben, kann diese Berechnung sehr aufwendig werden. Je mehr Objekte hinzugefügt werden, desto öfter müssen alle Elemente neu berechnet werden. Das selbe Problem tritt bei Objekten vom Typ HashSet auf, da diese intern auf eine HashMap zurückgreifen. Bei ArrayList Objekten wird in ähnlicher Weise der gesamte alte Array in den neuen kopiert.

Zur Optimierung dieser Vorgänge wurde der Code nun angepasst, um in möglichst vielen Fällen bereits beim Erstellen der HashMap eine zu erwartende Größe anzugeben. So reserviert die HashMap direkt ausreichend Speicher, um seltener die eigene Größe anzupassen. In den Messungen ist anschließend eine Verbesserung der Laufzeit zu erkennen.

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	00:35 min	00:41 min	00:40 min	00:39 min
GridSearch	00:30 min	00:30 min	00:30 min	00:30 min
Convolution	00:06 min	00:04 min	00:04 min	00:05 min

Tabelle 4.4: Zeitmessung: HashMap-Presizing

4.4 Optimierungsergebnisse

Im Vergleich zu den Messungen vor den beschriebenen Optimierungen ist ein deutlicher Performance-Gewinn erkennbar. Die Ausführungszeit wurde in den Beispiel-Fällen je nach Hardware stark reduziert. Beim Matrix-Beispiel und dem GridSearch-Beispiel sinkt die Ausführungszeit um etwa 20-25%, im Falle des Convolution-Beispiels sogar auf etwa 7% der ursprünglichen Ausführungs-dauer. Im Flamegraph und der Heatmap sind deutliche Verbesserungen zu erkennen. So nimmt nun das Kopieren von Arrays und das Konvertieren von Objekten zu Strings nur noch kleine Teile der Gesamtlaufzeit ein. Die Verwendung von HashMaps sinkt ebenfalls und das vorherrschende Kriterium stellt nun die ebenfalls kürzere Garbage-Collection dar.

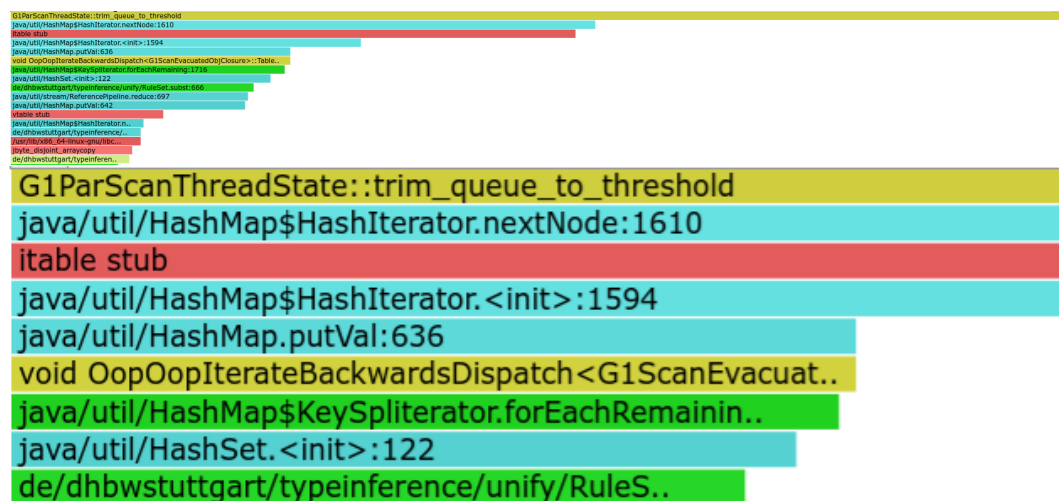


Abbildung 4.10: Heatmap mit allen Optimierungen für das Matrix-Beispiel

Kapitel 5

Umsetzung des Webservice

5.1 Server-Konzeption

Für die Umsetzung eines Servers gibt es einige Entscheidungen, die als Leitfaden für die Umsetzung getroffen werden mussten. Zuerst muss entschieden werden, ob der Server als eigenständiges Programm, als Erweiterung des Compilers oder in den Compiler integriert werden soll.

- Würde der Server als eigenständiges Programm umgesetzt werden, könnte dieser den benötigten Code selbst implementieren und so sehr spezialisiert auf interne Prozesse funktionieren. Es würde sich allerdings jeglicher Arbeitsaufwand doppeln, um beide Programme aktuell und einheitlich zu halten.
- Der Server könnte als Erweiterung des Compilers implementiert werden und diesen als einen Sub-Prozess zur Berechnung starten. Damit könnte nicht nur Compiler-Logik von Server-Logik getrennt werden, auch einzelne Berechnungen könnten weiterhin mit dem vorhandenen Code ablaufen. Da jede Berechnung in einem einzelnen Prozess abläuft, müsste kein Code für Mehrfachausführung angepasst werden. Jedoch müsste so eine zusätzliche Schnittstelle zwischen Server und Compiler geschaffen werden und immer beide Programme zur Verfügung stehen.
- Letztlich wurde sich für eine Integrierung des Servers in den Compiler entschieden. Dabei sind zwar Anpassungen am Compiler-Code nötig, aber im Gegenzug bieten sich mehrere Vorteile: Klassen und Logik können einfach wieder verwendet werden, Anpassungen gelten gleichzeitig für Server und Compiler und der gesamte Code ist in einem Projekt enthalten.

Im zweiten Schritt sind die unterliegenden Protokolle des Servers relevant. Da eine verlustfreie und geordnete Übertragung für einen erfolgreichen Einsatz wichtig ist, sollte auf dem Transport-Layer TCP (Transmission Control Protocol) gearbeitet werden. Durch UDP (User Datagram Protocol) wären Übertragungsfehler oder Datenverluste möglich, was in diesem Fall ein Problem darstellt.

Darauf aufbauend kann nun ein Anwendungsprotokoll gewählt werden. Hierbei stehen mehrere Optionen zur Verfügung:

- HTTP/2 (Long-)Polling stellt wiederholt Anfragen an den Server. Hierbei ist bei jeder Anfrage eine neue Verbindung nötig. Das erzeugt Overhead und bietet dem Server nur eingeschränkt die Möglichkeit, Nachrichten an den Client zu senden. Diese Option kommt oft bei Web-Anwendungen als Zwischenlösung vor, bietet aber für diesen Fall keine nennenswerten Vorteile.
- WebSockets bauen eine Verbindung per HTTP auf, wechseln dann aber in das eigene Protokoll und erhalten so eine andauernde Verbindung mit bidirektionaler Kommunikation. Für die Umsetzung des Web-Services bieten WebSockets eine einfach erweiterbare Grundlage, ohne viel Overhead.
- Ebenso existieren noch Erweiterungen der Konzepte HTTP-Polling und WebSockets, welche je einige Aspekte der Übertragung verbessern. Einige Beispiele dafür sind:
 - SOAP (Simple Object Access Protocol), welches auf HTTP-Rest Schnittstellen aufbaut und genaue Datenstrukturen definiert.
 - gRPC (Google Remote Procedure Call), über das eine WebSocket Verbindung für ein Service-Framework verwendet wird. Server-Code kann durch solche Services beinahe nahtlos aufgerufen werden. Automatisch generierter Code, sowie die binäre Übertragung von Daten bieten allerdings keine gute Grundlage für Fehlersuche und Debugging.

Als einfachste und praktischste Variante wurde sich daher für WebSockets entschieden. Für die Übertragung der Daten stehen wiederum mehrere Datenformate zur Verfügung, von denen drei hier aufgeführt werden:

- Binäre Darstellung von Objekten. Diese ermöglicht ein sehr kompaktes Datenformat, ist allerdings durch die Unleserlichkeit schwerer zu debuggen. Ein Textbasiertes Format wird daher bevorzugt.
- Textbasiert mit XML. Mittels XML wären genaue Schema-Vorgaben zur Übertragung möglich. Die Nutzung von Tags und Attributen erlaubt

außerdem eine kompaktere Darstellung von vielschichtigen Daten. Durch die Syntax entstehen allerdings große Text-Blöcke.

- Textbasiert mit JSON. Als häufiger Standard bietet JSON neben einer umfassenden Integration in diversen Programmiersprachen auch ein kompakteres Format als XML mit vergleichbarer Lesbarkeit. Durch die Vereinfachung zu Objekten, Listen und primitiven Literalen geht dabei allerdings viel Typ-Information verloren, welche über zusätzliche Daten angehängt werden muss.

Da bei dieser Umsetzung nur wenige Anfragen mit mittelmäßiger Datenmenge versendet werden sind diese Nachteile von JSON vernachlässigbar. Der zusätzliche Zeitaufwand zur Serialisierung und Übertragung fällt im Kontext der aufwendigen Berechnung der Unifikation ebenfalls nicht auf.

5.2 Vorbereitungen

Um aus dem lokalen Programm eine teilweise serverseitige Anwendung zu machen, werden einige Änderungen und Neuerungen benötigt. Diese lassen sich in zwei Kategorien einteilen. Zum einen gibt es Anpassungen am bestehenden Code als Voraussetzung für eine mehrfache Ausführung des Algorithmus:

- Keine Seiteneffekte im serverseitigen Code, der für den Client ausgeführt wird. Gibt es hierbei einen Seiteneffekt, kann dieser zukünftige und unabhängige Berechnungen beeinflussen. hierauf wird in Abschnitt 5.2.1 eingegangen.
- Die relevanten Teile des Programms müssen möglichst unabhängig vom übrigen Codes ausgeführt werden können. Jede zu Beginn der Verbindung existierende Abhängigkeit muss an den Server übermittelt werden und alle Ergebnisse müssen zurück an den Client übertragen werden. Daher muss versucht werden, die zu übertragenden Daten so kompakt wie möglich zu halten. Dieses Problem wird in Abschnitt 5.3.3 behandelt.

Zudem muss der Server und die Verbindung zu diesem implementiert werden.

- Es muss eine Server-Implementierung existieren, die unabhängig von der clientseitigen Anwendung des Compilers gestartet werden kann. Für die Wiederverwendbarkeit von Definitionen auf beiden Seiten wird dieser in Abschnitt 5.3.1 direkt mittels eines zusätzlichen Kommandozeilen-Arguments eingebunden.

- Es muss client- und serverseitig eine Kommunikationsmöglichkeit definiert und etabliert werden. Da potenziell mehrere Nachrichten asynchron von beiden Seiten aus gesendet werden sollen, wurde sich hier für eine Websocket-Verbindung entschieden.
- Um komplexere Daten über die Verbindung senden zu können, wird ein einheitliches Format benötigt, in welchem diese serialisiert und vom Empfänger zurück in Objekte konvertiert werden können. Als gängiges und textbasiertes Daten-Format für das Web wurde JSON gewählt und in Abschnitt 5.3.2 implementiert.
- Zur Vermeidung von fest im Code hinterlegten Routinen, wird zudem in Abschnitt 5.3.3 ein Paket-System für die Kommunikation zwischen Server und Client eingeführt. Dieses macht die Verbindung in Zukunft einfach erweiterbar und erlaubt das Kapseln von Daten und Code zur Verarbeitung in einzelnen spezialisierten Klassen.

Abschließend werden Unit-Tests für die neue Implementierung erstellt, um auch bei zukünftigen Änderungen die Funktionen zu prüfen.

5.2.1 Kapselung des Unifikations-Algorithmus

Der Compiler war bisher nur auf eine einzige Berechnung pro Ausführung ausgelegt. Das zeigt sich vor allem in der Verwendung von statischen Variablen und der häufigen Wiederverwendung von Code zwischen Packages. Für einen Server sind solche Verbindungen allerdings problematisch, da sie Abhängigkeiten zwischen Anfragen erzeugen. Ergebnisse können dadurch verändert oder sogar falsch berechnet werden, sollten noch Daten von einem vorherigen Durchlauf berücksichtigt werden. Damit die Unifikation auf einen Server ausgelagert werden kann, müssen die verwendeten Funktionen rein sein. Eine Funktion gilt als “rein”, wenn sie für identische Argumente auch identische Ausgaben erzeugt. Problematisch sind dafür die Verwendung statischer Variablen, das mutieren von mehrfach verwendeten Objekten und jegliche Form von nicht-deterministischem Verhalten. Beispielsweise Race-Conditions bei Threads oder Zufallsgeneratoren. Im Folgenden werden die wichtigsten Änderungen zum Erreichen dieser Bedingungen aufgeführt.

5.2.2 Kapselung der Platzhalter Generierung

Die bisherige Implementierung von Platzhalter-Namen in der Klasse “PlaceholderType” verwendet eine statische Variable zum Speichern der generierten Platzhalter. Bei einem Server, welche viele unabhängige Vorgänge im selben Prozess verarbeitet, würde das für einen problematischen Seiteneffekt sorgen.

Es könnten vom Client generierte Platzhalter erneut angelegt werden, oder aber die Platzhalter mit jedem Aufruf immer länger werden, um nicht mit alten Platzhaltern in Konflikt zu geraten.

Dieses Problem wurde bereits in Abschnitt 4.3.2 teilweise gelöst, aber noch nicht vollständig behoben. Daher wurde nun zusätzlich die Platzhalter-Generierung und Speicherung in eine eigene Klasse “PlaceholderRegistry” verschoben. Dieses Objekt kann nicht nur für jede Ausführung des Codes individuell neu erzeugt werden, sondern erlaubt auch das Synchronisieren von Client-Platzhaltern und Server-Platzhaltern durch Übertragung der Daten. So bleibt die Platzhalter-Generierung gekapselt.

Ein weiteres Problem in diesem Kontext ist die Generierung von Platzhalter-Namen bei der Erstellung des AST über eine separate Klasse “NameGenerator”. Dessen generierte Platzhalter sind der “PlaceholderRegistry” nicht bekannt. Dieses Problem lässt sich aber dadurch recht einfach lösen, dass jeder Aufruf der NameGenerator Klasse den generierten Namen auch zur PlaceholderRegistry hinzufügt. Da diese noch immer auf die Existenz von neu generierten Namen prüft, können so Doppelungen ausgeschlossen werden.

5.2.3 Kapselung von Parametern

Im Unifikations-Algorithmus müssen zahlreiche Parameter über weite Teile des Codes von Methode zu Methode übergeben werden. Das beinhaltet zum Beispiel einen booleschen Parameter “parallel”, der zwischen paralleler oder serieller Ausführung unterscheidet. Aber auch ein Writer Objekt für die Ausgabe in Dateien und einen dazugehörigen “log” Wert zum Ein- und Ausschalten des Datei-Outputs, werden mit übergeben. Zusätzlich existieren statische Writer-Objekte, die ebenfalls ein Problem für die Kapselung darstellen.

Beide Fälle wurden durch die Einführung einer neuen Klasse gelöst, den “UnifyContext” (siehe Abb. 5.1). Dieser Kontext enthält allgemeine Daten über eine Unifikation und kann anstelle vieler einzelner Parameter von Methode zu Methode durchgereicht werden. Damit wird die Anzahl der Methoden- und Klassen-Parametern in der Unifikation verringert. Gleichzeitig bietet sich nun ein einfacher Weg, um ohne große Anpassungen Objekte wie Log-Writer oder die zuvor erstellte “PlaceholderRegistry” im Algorithmus zu übergeben. Da sich Parameter in tieferen Ebenen des Algorithmus auch ändern können, müssen ebenfalls Änderungen am UnifyContext erlaubt sein. Um aber vorherige Kontexte nicht zeitgleich zu verändern, muss jeder UnifyContext unveränderlich (“immutable”) sein. So kann sich eine Methode darauf verlassen, dass sich der übergebene Kontext nie ändert. Im UnifyContext implementieren wir daher Methoden, die einen neuen UnifyContext mit einzelnen Änderungen erzeugen und so alle identischen Parameter beibehalten.

```
1 public record UnifyContext(  
2     Logger logger,  
3     boolean parallel,  
4     PlaceholderRegistry phRegistry,  
5     ...  
6 ) {  
7     // Änderungen ausschließlich über die Erzeugung neuer Instanzen  
8     public UnifyContext setParallel(boolean parallel) {  
9         if (this.parallel == parallel) return this;  
10        return new UnifyContext(logger, parallel, phRegistry, ...);  
11    }  
12 }
```

Abbildung 5.1: Vereinfachte Version der Unify-Context Klasse

5.2.4 Implementierung eines Loggers

Ausgaben und Log-Dateien werden im Compiler bisher ad hoc umgesetzt. Wo eine Ausgabe gemacht werden soll, wird ein “System.out.println(...)” platziert und wo in eine Datei geschrieben werden soll, wird repetitiv ein ganzer mehrzeiliger Code-Block mit einer Abfrage des log-Wertes, einem try-catch, beliebig vielen Schreib-Aufträgen und einem Error-Handling (ebenfalls über “System.out.println”) eingefügt. Das beeinträchtigt die Leserlichkeit des Codes und verhindert auch selektive Ausgaben.

Zudem wäre auf Seite des Servers sowohl das Erzeugen von Log-Dateien aufgrund endlichen Speicherplatzes, als auch das Ausgeben von zu viel Output hinderlich. Für Ausgaben auf den Standard-Output-Stream wäre eine Weiterleitung an den Client nützlich.

Daher wurde eine einheitliche Klasse “Logger” implementiert. Diese kann je nach Bedarf mit oder ohne Writer-Objekt und einem eindeutigen Präfix erzeugt werden. Sämtliche manuelle Ausgaben wurden zu Aufrufen dieser Klasse geändert und bei Gelegenheit ein Präfix zugewiesen. Somit wird das gesamte Output-System vereinheitlicht und kann gezielt angepasst werden, während Ausgaben mit ihrem Teilbereich als Präfix erscheinen.

Die bisher im Code fest implementierte Eigenschaft “log” (für Datei-Ausgaben) wurde mitsamt einem neuen Feature in die Kommandozeilen-Argumente des Programms übernommen. Beim Start kann damit nicht nur die Ausgabe

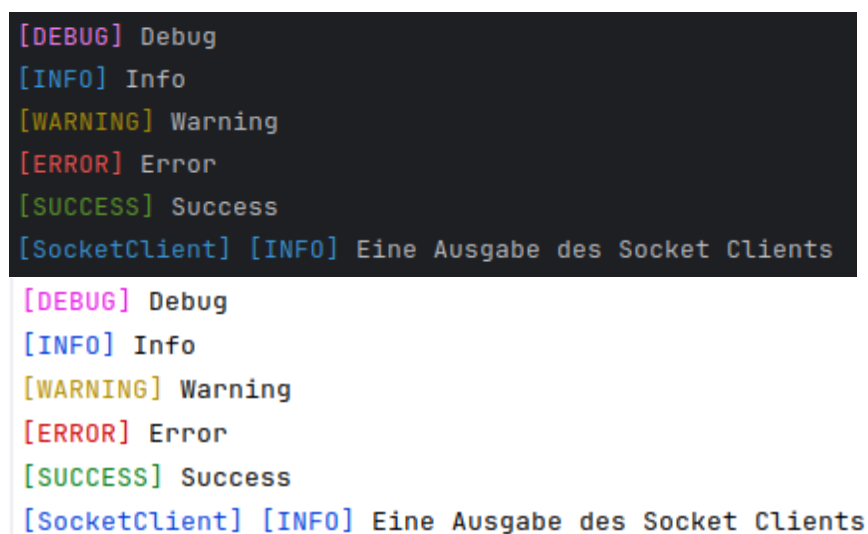
in Dateien ohne Code-Änderung aktiviert werden, es sind zudem mehrere Level an Ausgabe-Niveau möglich. Während mittels `-vvv` sämtliche Ausgaben angezeigt werden, werden zum Beispiel mit `-v` nur die Ausgaben mit Level `warning` und höher angezeigt. Insgesamt gibt es fünf Level (`debug`, `info`, `warning`, `error`, `success`), die allein durch den Methodenaufruf geändert werden können. Für eine bessere visuelle Trennung von Ausgaben wird das Prefix und das Ausgabe-Level zudem mittels der Bibliothek [Nun22] passend farblich hervorgehoben. Dabei verwendet der Logger ausschließlich ANSI-Escapesequenzen, die moderne Systeme im Normalfall unterstützten. Dadurch werden aufeinanderfolgende Ausgaben besser voneinander unterscheidbar.

```

1 // Angenommen, das Programm wurde mit -v (Warning) gestartet
2 logger.debug("Debug");           // Ausgabe:
3 logger.info("Info");             // Ausgabe:
4 logger.warn("Warning");          // Ausgabe: [WARN] Warning
5 logger.error("Error");           // Ausgabe: [ERROR] Error
6 logger.success("Success");       // Ausgabe: [SUCCESS] Success
7 logger.exception(new RuntimeException("message"));
8     // Ausgabe: [ERROR] Exception "message"
9     //           | StackTrace1
10    //           | StackTrace2 ...

```

Abbildung 5.2: Logger Nutzung und Ausgaben



```

[DEBUG] Debug
[INFO] Info
[WARNING] Warning
[ERROR] Error
[SUCCESS] Success
[SocketClient] [INFO] Eine Ausgabe des Socket Clients

[DEBUG] Debug
[INFO] Info
[WARNING] Warning
[ERROR] Error
[SUCCESS] Success
[SocketClient] [INFO] Eine Ausgabe des Socket Clients

```

Abbildung 5.3: Farb-Schema des Loggers

Für den Server wurde zudem eine Erweiterung des Loggers implementiert, die anstelle von Ausgaben auf dem Terminal alle Ausgaben über Pakete an den Client sendet und dessen Ausgabe-Level übernimmt.

5.3 Implementierung des WebSocket-Servers

Die Implementierung eines WebSocket-Servers erfordert zuerst zwei neue Programm-Argumente:

- Mit dem Argument `--server-mode <port>` startet das Programm anstelle des normalen Compilers einen Server, welcher auf dem angegebenen Port auf eingehende Verbindungen akzeptiert.
- Mit dem Argument `--unify-server <url>` wird dem Java-TX-Compiler signalisiert, bei Bedarf den Server unter der angegebenen URL anzufragen. Da es sich um einen WebSocket Server handeln wird, muss die URL auch dieses Protokoll widerspiegeln. Zum Beispiel `“ws://localhost:5000”`, falls der Server auf dem selben Gerät und auf Port 5000 läuft.

Die Unify-Server URL wird als `Optional<String>` an die Haupt-Klasse `JavaTXCompiler` übergeben. Für den Server wurde eine neue Klasse im `“core”` package namens `JavaTXServer` erstellt. Wird das Programm mit dem Server-Mode Argument aufgerufen, wird anstelle des `JavaTXCompiler` Objekts ein `JavaTXServer` Objekt erzeugt und an dieses der entsprechende Port übergeben. Die beiden Argumente schließen sich gegenseitig aus. Somit bricht das Programm mit einem Fehler ab, sollte dieser Fall eintreten.

Damit steht nun die nötige Infrastruktur für die Implementierung der neuen Verbindung auf Client- und Serverseite zur Verfügung. Da sich beide Seiten im selben Projekt befinden, können auch alle Klassen in beiden Fällen genutzt werden.

5.3.1 Server-Endpoint Implementierung

Die Implementierung des WebSocket-Servers verwendet die Bibliothek “Java Websockets” [Raj24]. Im Vergleich zu einer Eigenentwicklung stellt die Bibliothek zentrale Funktionen bereits fertig zur Verfügung. Das beinhaltet unter anderem:

- Bereits implementierte Funktionen für Handshakes, Message-Framing und Pings/Pongs (als Regelmäßiger Verbindungsscheck).
- Es werden Callback Methoden für alle gängigen Events bereitgestellt.
- Thread-Management für eingehende Verbindungen.
- Kompression über PerMessageDeflation.
- Erweiterbarkeit für SSL/TSL, was im Folgenden noch nicht genutzt wird.

Die gängigen Standards für Websockets und Server werden von der Bibliothek vollständig erfüllt. Das umfasst vor allem die Definitionen [AM11] und [Yos11].

Verbindungsaufbau

Beim ersten Einsatz des Socket-Clients wird die Verbindung zum Server hergestellt. Dabei wurde eine zusätzliche Header-Information hinzugefügt, die eine feste Paket-Protokoll-Version enthält.

```
1 public static final String packetProtocolVersion = "1";
```

Stimmt der Wert im Header nicht mit dem Wert auf dem Server überein, so wird die Verbindung vom Server abgebrochen. Somit ist es nur bei einer Server-Client Kombination mit identischer Versionsnummer möglich, miteinander zu kommunizieren und Fehler durch unvollständige Daten werden umgangen.

Die WebSocket-Bibliothek übernimmt das regelmäßige Senden von Ping-Signalen zwischen Server und Client, auf die die Gegenseite mit einem Pong-Signal antwortet. Sollte eine Seite für eine gewisse Zeit kein Antwort-Signal erhalten, so wird von einem Verbindungsabbruch oder Timeout ausgegangen und der WebSocket geschlossen.

Im normalen Verlauf wird die Verbindung geschlossen, sobald eine der beiden Seiten beendet wird. Um hier bei lange dauernden Berechnungen die Verbindung vorzeitig schließen zu können, wurde ein zusätzliches “Auto-Close” Signal eingeführt. Wurde dieses vom Server empfangen, schließt dieser die Verbindung nach dem Erledigen des letzten offenen Tasks.

5.3.2 Daten-Serialisierung zu JSON

Für die Übertragung der Daten wurde sich für das Datenformat JSON entschieden. Für das Konvertieren zu und von JSON wurde die Bibliothek Jackson [Fas24] verwendet. Durch die Verwendung von Jackson-Annotations können beliebige Klassen für die Serialisierung konfiguriert werden und es wird eine Eigenentwicklung eines JSON-Parsers vermieden. Allerdings wurde zur besseren Trennung von Serialisierungs-Logik und Programm, sowie der einfacheren Handhabung des Typ-Verlusts bei der Serialisierung eine Menge von Klassen eingeführt, welche als Zwischenschritt zur Serialisierung und zurück funktionieren. Somit können auch die eigentlichen Klassen unverändert bleiben, da Jackson zum Beispiel einen Standard-Konstruktor benötigt, welcher in der Praxis oft nicht gewünscht ist. Zudem implementieren diese Klassen auch Methoden für den sicheren Zugriff auf die enthaltenen Daten.

- “SerialMap” erweitert die traditionelle HashMap und erlaubt das speichern von String-Object Paaren, wobei das Objekt eine der `Serial[...]` Klassen sein muss. Diese Klasse wird im JSON als Objekt dargestellt.
- “SerialList” analog zur SerialMap erweitert diese Klasse die ArrayList und wird im JSON als Array dargestellt.
- “SerialValue” erlaubt das einbinden von einzelnen Werte, die entweder einen primitiven Typ oder den Typ String besitzen.
- “SerialUUID” stellt einen Spezialfall des SerialValue dar. Da einige Datenstrukturen rekursiv auftreten können, werden diese mit einer UUID versehen und einmalig in einer gesonderten SerialMap angelegt. Bei der Deserialisierung kann über die UUID wieder auf das Element der SerialMap zugegriffen werden. Recursive Verbindungen bleiben somit erhalten.

Diese Klassen erben von einer gemeinsamen Elternklasse “ISerialNode”, welche die für Jackson benötigten Mapping-Informationen enthält. Wie in Abb. 5.4 zu sehen, werden die zuvor erstellten Klassen als mögliche Typen definiert und erhalten je einen eindeutigen Namen zur Identifikation. Dieser Name wird als Eigenschaft “_t” an Objekte angehängt, um den korrekten Typ für die Rekonstruktion zu speichern.

```
1 import com.fasterxml.jackson.annotation.JsonSubTypes;
2 import com.fasterxml.jackson.annotation.JsonTypeInfo;
3
4 @JsonTypeInfo(
5     use = JsonTypeInfo.Id.NAME,
6     include = JsonTypeInfo.As.PROPERTY,
7     property = "_t"
8 )
9 @JsonSubTypes({
10     @JsonSubTypes.Type(value = SerialMap.class, name = "m"),
11     @JsonSubTypes.Type(value = SerialList.class, name = "l"),
12     @JsonSubTypes.Type(value = SerialValue.class, name = "v"),
13     @JsonSubTypes.Type(value = SerialUUID.class, name = "u")
14 })
15 public interface ISerialNode {
16     ...
17 }
```

Abbildung 5.4: Interface: ISerialNode

Mit einem neuen Interface `ISerializableData` können Objekte die Methoden eine `toSerial` und eine statische `fromSerial` Methode einbinden (siehe Anhang B.1).

Um auch rekursive Beziehungen darzustellen und später mit korrekter Referenz wiederherstellen zu können, wird bei der Serialisierung ein Objekt der Klasse `KeyStorage` übergeben. Über diesen kann eine Serielle Darstellung hinter einer UUID abgelegt und später abgefragt werden. Somit kann ein Objekt mehrfach referenziert werden und wird höchstens einmal verarbeitet.

Eine Beispiel-Implementierung der der Serialisierung und Deserialisierung anhand der Klasse `Pair` ist in Anhang B.2 zu finden. Durch den Aufruf der Jackson-Bibliothek, kann anschließend aus den serialisierten Klassen das finale JSON-Konstrukt erzeugt werden.

```

1  {   "_t": "m",
2      "op":      { "_t": "v", "value": "=." },
3      "noUnification": { "_t": "v", "value": 0 },
4      "ta2": { "_t": "m",
5              "type": { "_t": "v", "value": "RefType" },
6              "object": { "_t": "m",
7                          "name": { "_t": "v", "value": "java.lang.Integer" },
8                          "isPrimitive": { "_t": "v", "value": false },
9                          "parameters": ["1", []]
10             }
11     },
12     "ta1": { "_t": "m",
13             "type": { "_t": "v", "value": "TypePlaceholder" },
14             "object": { "_t": "m",
15                         "wildcardable": { "_t": "v", "value": true },
16                         "variance": { "_t": "v", "value": 0 },
17                         "name": { "_t": "v", "value": "Q" }
18             }
19     },
20     "location": { "_t": "m",
21                 "file": { "_t": "v", "value": "Op.jav" },
22                 "line": { "_t": "v", "value": 14 }
23     }
24 }

```

Abbildung 5.5: Beispiel: JSON Darstellung eines Pair-Objekts

Die entstehenden JSON Daten erscheinen zuerst relativ groß für die Menge an enthaltenen Daten. Durch die Anwendung der Typ-Informationen durch Jackson wird das noch verstärkt. Allerdings kann die Datenmenge durch die Anwendung von PerMessageDeflation [Yos11] bei der Übertragung deutlich verringert werden. Dabei wird mittels den Algorithmen LZ77 und Huffman-Encoding zuerst nach häufigen Zeichenketten und dann häufigen Zeichen komprimiert. Regelmäßige Strukturen wie in diesem Fall profitieren von dieser Methode. Die Schlüssel in Objekten könnten zudem noch weiter verkürzt werden, was aber zum Zwecke der Lesbarkeit hier nicht umgesetzt wurde.

```
1 {
2   "op": "=",
3   "noUnification": false,
4   "ta2": {
5     "type": "RefType",
6     "object": {
7       "name": "java.lang.Integer",
8       "isPrimitive": false,
9       "parameters": []
10    }
11  },
12  "ta1": {
13    "type": "TypePlaceholder",
14    "object": {
15      "wildcardable": true,
16      "variance": 0,
17      "name": "Q"
18    }
19  },
20  "location": {
21    "file": "Op.jav",
22    "line": 14
23  }
24 }
```

Abbildung 5.6: Beispiel: JSON Darstellung der Klasse Pair

Aus dem Beispiel ist zu erkennen, dass es sich bei dem serialisierten Objekt um folgendes Paar handelt: `java.lang.Integer =.` TPH Q, das aus der Datei `Op.jav` Zeile 14 generiert wurde.

5.3.3 Implementierung eines Packet-Systems

Da nun eine Verbindung zwischen Server und Client existiert und wir über JSON Daten austauschen können, ist ein einheitliches Format für das austauschen von Informationen erforderlich. Dafür werden drei Interfaces implementiert:

```
1 public interface IPacket {}
2
3 public interface IClientToServerPacket extends IPacket {
4     @JsonIgnore
5     void onHandle(WebSocket websocket, SocketServer socketServer);
6 }
7
8 public interface IServerToClientPacket extends IPacket {
9     @JsonIgnore
10    void onHandle(WebSocket websocket, SocketClient socketClient);
11 }
```

Abbildung 5.7: Packet-System Interfaces

Über diese Interfaces können Daten-Pakete erstellt werden. Dabei muss jedes Paket eine Handler-Methode implementieren, die von der Empfängerseite ausgeführt wird. Als Beispiel ist in Anhang B.3 die Implementierung einer Klasse “MessagePaket” zu finden, welches die jeweils andere Seite zur Ausgabe eines Textes auffordern kann.

5.4 Ergebnis der Server-Implementierung

Durch die Implementierung des WebSocket-Servers konnte die Unifizierung von Java-TX auf eine flexible Client-Server-Architektur umgestellt werden. Der Server übernimmt dabei Teile der rechenintensive Verarbeitung, während der Client lediglich die Anfragen stellt und Ergebnisse empfängt. Die gewählte Struktur mit JSON-Serialisierung, fest definiertem Packet-System und automatischem Verbindungs-Handling erlaubt eine saubere Trennung zwischen Datenübertragung und eigentlicher Compiler-Logik.

In Abb. 5.8 ist beispielhaft eine Übersicht der Kommunikation zwischen Client und Server während der Nutzung des Web-Service zu sehen. Dabei fragt der Client beim Server eine Unifikation an und fordert diesen zum Schließen der Verbindung nach Erledigung des Tasks auf. Während der Verarbeitung sendet der Server die Ausgaben zum Client, gefolgt von dem Ergebnis der Unifikation. Abschließend wird die Verbindung vom Server geschlossen.

Je nach Verlauf kann das tatsächliche Verhalten von dem dargestellten Beispiel abweichen. Im Falle eines Fehlers würde kein Unifikations-Ergebnis, sondern ein Fehler-Paket gesendet. Oder der Client könnte auf das Auto-Close Paket verzichten und die Verbindung bei Ende der eigenen Ausführung selbst schließen. Sollte außerdem bis 10 Sekunden nach Öffnung der Verbindung keine relevante Anfrage beim Server eingehen, schließt dieser automatisch die Verbindung.

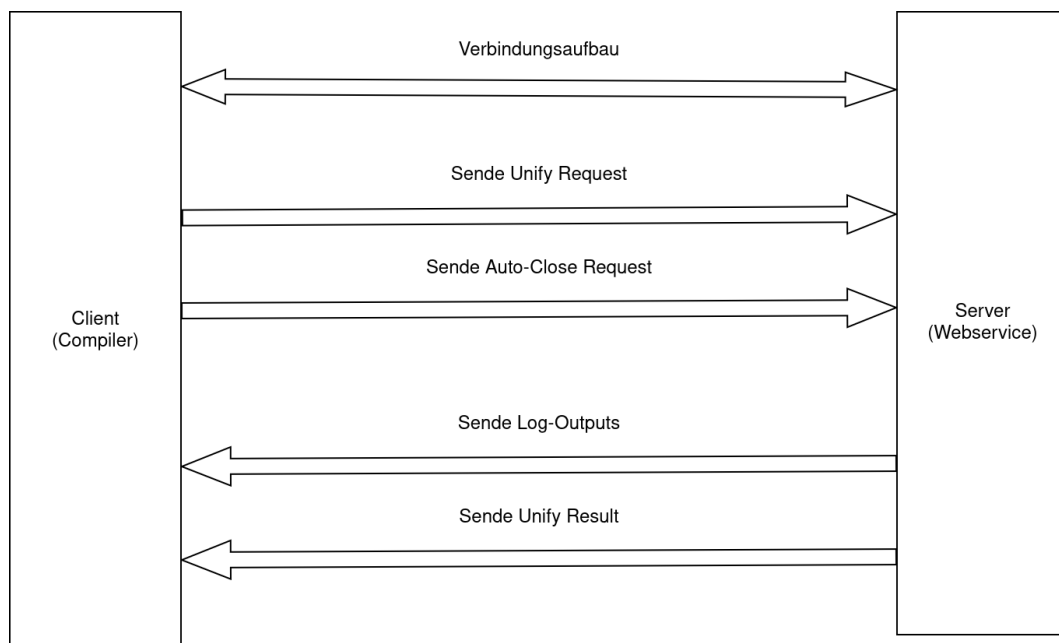


Abbildung 5.8: Webservice Ablauf

5.4.1 Zeitmessung mit Webservice

Um den tatsächlichen Performance-Gewinn durch die Umsetzung festzustellen, wurden weitere Tests gemacht. Diese bieten einen Vergleichswert zwischen der lokalen und der serverseitigen Ausführung. Um Performance-Unterschiede besser zuordnen zu können, wurden die Performance-Tests zusätzlich auch rein serverseitig ausgeführt.

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	03:16 min	03:20 min	03:28 min	03:21 min
GridSearch	03:42 min	03:46 min	03:42 min	03:53 min
Convolution	03:06 min	03:08 min	03:16 min	03:10 min

Tabelle 5.1: Zeitmessung: Master-Branch auf dem Test-Server

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	00:50 min	00:53 min	00:51 min	00:51 min
GridSearch	01:00 min	00:55 min	00:57 min	00:57 min
Convolution	00:10 min	00:09 min	00:09 min	00:09 min

Tabelle 5.2: Zeitmessung: Optimierter Code auf dem Test-Server

	1. Durchlauf	2. Durchlauf	3. Durchlauf	Durchschnitt
Matrix	00:35 min	00:49 min	00:40 min	00:41 min
GridSearch	00:57 min	00:56 min	00:56 min	00:56 min
Convolution	00:12 min	00:08 min	00:08 min	00:08 min

Tabelle 5.3: Zeitmessung: Lokale Ausführung mit Webservice

Aus den gemessenen Zeiten in den Tabellen 5.1 und 5.2 kann auch hier eine deutliche Steigerung der Performance im Vergleich zum master-Branch heraus gelesen werden. Allerdings liegt dieser noch immer hinter den Messwerten bei der lokalen Ausführung. Dies weist nochmal auf eine starke Abhängigkeit des Programms von der ausführenden CPU hin, wohingegen die Stärke des Servers auf einer hohen Anzahl Threads basiert.

Für ältere oder schwächere Geräte kann die Nutzung des Servers dennoch einen tatsächlichen Performance-Boost bedeuten und die Ausführung unabhängiger von der lokalen Hardware machen. So würde ein Gerät mit schwacher CPU und wenigen Kernen ebenfalls die Performance des Servers aufweisen, wie in Tabelle 5.3 gemessen wurde. Bei stärkeren lokalen Geräten ist allerdings kein Vorteil durch die Server-Nutzung zu erwarten.

Kapitel 6

Tests - Abläufe und Ergebnisse

Durch die Erstellung von Tests soll die neue Server-Funktionalität sichergestellt und Probleme bei der Übertragung vermieden werden. Auch zukünftige Änderungen profitieren von diesen Tests, da entstehende Problematiken direkt erkannt werden.

6.1 Unit-Tests

Es wurden Unit-Tests für die Serialisierung von Daten nach JSON und zurück erstellt. Dabei darf es weder zu Daten-Verlusten, noch zu Änderungen an den Daten oder deren Beziehungen kommen. Dafür wird die Konvertierung in Hin- und Rückrichtung für Beispiel-Daten und anschließend ein direkter Vergleich durchgeführt. Ein Beispiel-Test ist in Anhang C.1 zu finden.

6.2 Integrations-Tests

Zur Vereinfachung wurden die folgenden Integrations-Tests über JUnit als Unit-Tests implementiert.

Für das sicherstellen einer identischen Verarbeitung von Daten auf Server und Client wird mittels einem Integrationstest dieses Verhalten simuliert und beide Ergebnisse verglichen. Dafür wird ein lokaler Server auf Port 5000 gestartet und Beispieldaten erzeugt. Für diese Beispiel-Daten wurde anschließend die Typ-Inferenz über den Server und danach direkt ausgeführt. Der entstehende Bytecode wird abschließend auf Gleichheit geprüft.

Kapitel 7

Zusammenfassung

Dieses Kapitel fasst die Arbeit zusammen und stellt dann die erzielten Ergebnisse dar.

Zu Beginn wurden in Kapitel 2 grundlegende Konzepte zum vorliegenden Projekt um Java-TX und die Verwendung von Client-Server Strukturen beschrieben. Als Referenz für die beschriebenen Änderungen wurde in Kapitel 3 der verwendete Algorithmus beschrieben. In Vorbereitung für die Umsetzung des Client-Server Konstrukts wurden in Kapitel 4 einige Optimierungen getestet und durchgeführt. Darunter befinden sich zum Teil Anpassungen aus [Leh23], sowie neue Änderungen. Die Ausführungsdauer des Programms und insbesondere des Unifikations-Algorithmus konnten damit in den durchgeführten Tests auf mehr als 25% gesenkt werden.

Auf dieser Basis wurde im Anschluss in Kapitel 5 die Implementierung des Client-Server Konzepts vorgenommen. Mittels Websockets, JSON und einem eigenen Paket-Format wurde ein Webservice implementiert. Dieser ermöglicht es nun, eine beliebige Anzahl Tasks zur Laufzeit an den Server zu senden, dort bearbeiten und optional beantworten zu lassen. Somit wurde das primäre Ziel dieser Arbeit, die Umsetzung eines funktionierenden, effizienten und dynamischen Webservices, erreicht. Ob die Server-Implementierung auch einen praktischen Vorteil gegenüber der lokalen Ausführung bietet, wurde in Abschnitt 5.4.1 getestet und mit der lokalen Ausführung verglichen.

7.1 Ergebnis

Eine generelle Verbesserung der Performance durch die Webservice-Implementierung konnte nicht erreicht werden. Aufgrund der Abhängigkeit des Programms von der CPU, dem Multi-Threading-Overhead und der Garbage-Collection ist der Vorteil einer hohen Zahl an Threads eher gering. Die Performance-Optimierungen bieten jedoch auch für die lokale Ausführung eine deutliche Verbesserung der Ausführungsdauer. Zudem kann der Unifikations-Server auch für ältere oder langsamere Geräte verwendet werden, womit zumindest bei diesen eine Steigerung der Performance zu erwarten ist. Im Allgemeinen wurden zudem Verbesserungen an der Leserlichkeit und Nutzung des Compilers vorgenommen, von denen auch zukünftige Arbeit am Code profitieren kann.

7.2 Ausblick

Mit der Umsetzung aus dieser Arbeit eröffnen sich durch die Umsetzung der Server-Anbindung mehrere mögliche Erweiterungen in der Zukunft. So wäre es auch möglich, mehrere unabhängige Tasks auf dem Server durch die hohe Thread-Zahl vollständig parallel ablaufen zu lassen. Eine weitere Möglichkeit wäre die Neu-Entwicklung des Unifikations-Algorithmus unter anderen Voraussetzungen, zum Beispiel einer nativ kompilierenden Programmiersprache, unabhängig vom Compiler-Projekt.

Die durchgeführten Verbesserungen an der Performance bieten dennoch bereits einen Vorteil gegenüber der vorherigen Umsetzung. Da alle Änderungen rein durch Optimierungen der Implementierung durchgeführt wurden, ergänzen diese sich mit potentiellen Optimierungen am Unifikations-Problem. Zum aktuellen Zeitpunkt werden an der DHBW unter anderem auch Tests zur Einschränkung der Finite-Closure durchgeführt, durch die in Kombination mit den Ergebnissen dieser Arbeit bereits eine Ausführungszeit der Beispiele im Bereich von unter drei Sekunden erreicht werden konnte.

Quellenverzeichnis

- [AM11] Ian Fette Alexey Melnikov. The websocket protocol, 2011.
<https://datatracker.ietf.org/doc/html/rfc6455>
Letzter Zugriff am 27. Juni 2025.
- [And25] Pangin Andrei, apangin. Async-profiler, 2025. Version 4.0
<https://github.com/async-profiler/async-profiler>
Letzter Zugriff am 13. Juli 2025.
- [DHBW25] Campus Horb Duale Hochschule Baden Württemberg, Stuttgart.
Java-tx compiler, 2025.
<https://gitea.hb.dhbw-stuttgart.de/JavaTX/JavaCompilerCore>
Letzter Zugriff am 30. Juli 2025.
- [Fas24] FasterXML. Jackson, 2024. Version 2.17.2
<https://github.com/FasterXML/jackson>
Letzter Zugriff am 05. Juli 2025.
- [Int17] ECMA International. Ecma-404 - the json data interchange syntax, 2nd edition, 2017.
<https://ecma-international.org/publications-and-standards/standards/ecma-404/>
Letzter Zugriff am 27. Juni 2025.
- [Leh23] Paul Lehmann. *Massiv paralleler Unifikationsalgorithmus*. Studienarbeit, Duale Hochschule Baden-Württemberg, Campus Horb, 2023.
- [Nun22] Diogo Nunes, dialex. Jcolor, 2022. Version 5.5.1
<https://github.com/dialex/JColor>
Letzter Zugriff am 05. Juli 2025.
- [Ora25] Oracle. Visualvm, 2025. Version 2.2
<https://visualvm.github.io/>
Letzter Zugriff am 13. Juli 2025.

- [Plü04] Martin Plümicke. Type Unification in Generic–Java. In *Proceedings of 18th International Workshop on Unification (UNIF'04)*, Cork, Ireland, 2004.
- [Plü09] Martin Plümicke. Java Type Unification with Wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *Applications of Declarative Programming and Knowledge Management*, pages 223–240. Springer Berlin Heidelberg, 2009.
- [Plü23] Martin Plümicke. Optimization of the Java Type Unification. In Sibylle Schwarz and Mario Wenzel, editors, *Proceedings of the 37th Workshop on (Constraint) Logic Programming (WLP 2023)*, 2023.
- [Raj24] Nathan Rajlich, TooTallNate. Java websockets, 2024. Version 1.5.2, <https://github.com/TooTallNate/Java-WebSocket>
Letzter Zugriff am 04. Juli 2025.
- [Sie89] Jörg H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, [https://doi.org/10.1016/S0747-7171\(89\)80012-4](https://doi.org/10.1016/S0747-7171(89)80012-4), 7(3):207–274, 1989. Unification: Part 1.
- [Ste15] Florian Steurer. *Implementierung eines Typunifikationsalgorithmus für Java 8*. Studienarbeit, Duale Hochschule Baden-Württemberg, Campus Horb, 2015.
- [Yos11] Takeshi Yoshino. Compression extensions for websocket, 2011. <https://datatracker.ietf.org/doc/html/rfc7692>
Letzter Zugriff am 14. Juli 2025.

Verwendung KI-gestützter Programme

Programm	LanguageTool (Firefox Browser-Extension)
Versionsnummer	9.0.1
Verwendungszweck	Rechtschreib- und einfache Grammatik-Prüfung
Verknüpfung	https://languagetool.org

Anhang A

Performance-Test Beispiele

A.1 Matrix.jav

```
1 import java.util.Vector;
2 import java.lang.Integer;
3
4 class Matrix extends Vector<Vector<Integer>> {
5     Integer mul1(Integer x, Integer y) { return x;}
6     Integer add1(Integer x, Integer y) { return x;}
7     mul(m) {
8         var ret = new Matrix();
9         var i = 0;
10        while(i < size()) {
11            var v1 = this.elementAt(i);
12            var v2 = new Vector<Integer>();
13            var j = 0;
14            while(j < v1.size()) {
15                var erg = 0;
16                var k = 0;
17                while(k < v1.size()) {
18                    erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
19                    k++; }
20                v2.addElement(new Integer(erg));
21                j++; }
22            ret.addElement(v2);
23            i++; }
24        return ret;
25    }
26 }
```

Abbildung A.1: Beispiel-Datei 1: Matrix.jav

A.2 GridSearch.jav

```
1  import java.lang.Integer;
2  import java.lang.Boolean;
3  import java.util.Queue;
4  import java.util.Vector;
5  import java.util.List;
6  import java.util.ArrayDeque;
7
8  class Pos {
9      public Integer x;
10     public Integer y;
11     public Pos(Integer x, Integer y) {
12         this.x = x; this.y = y;
13     }
14 }
15
16 class GridSearch {
17
18     Pos search(Vector<Vector<Boolean>> grid) {
19         var w = grid.size();
20         var h = grid.getFirst().size();
21         // keep a queue on which cells to check
22         var cellQueue = new ArrayDeque<Pos>();
23         cellQueue.add(new Pos(0,0));
24         while (!cellQueue.isEmpty()) {
25             var pos = cellQueue.poll();
26             // if the target was found: return the position
27             var value = grid.get(pos.x).get(pos.y);
28             if (value) { return pos; }
29             // keep searching on neighboring tiles
30             if (pos.x < w-1) cellQueue.add(new Pos(pos.x + 1, pos.y));
31             if (pos.y < h-1) cellQueue.add(new Pos(pos.x, pos.y + 1));
32         }
33         return null;
34     }
35 }
36 }
```

Abbildung A.2: Beispiel-Datei 2: GridSearch.jav

A.3 Convolution.jav

```
1 import java.util.ArrayList;
2 import java.util.Vector;
3 import java.lang.Integer;
4
5 class Pixel {
6     public color;
7 }
8
9 class Convolution {
10     mask;
11     Convolution(mask) {
12         this.mask = mask;
13     }
14     apply(pixels) {
15         var w = this.mask.size();
16         var h = this.mask.get(0).size();
17         var imgW = pixels.size();
18         var imgH = pixels.get(0).size();
19
20         for (var x = 0; x < imgW - w; x++) {
21             for (var y = 0; y < imgH - h; y++) {
22                 var total = 0;
23                 for (var xd = 0; xd < w; xd++) {
24                     for (var yd = 0; yd < h; yd++) {
25                         var p = pixels.get(x + xd).get(y + yd);
26                         var m = this.mask.get(xd).get(yd);
27                         total = total + (p.color * m);
28                     }
29                 }
30                 pixels.get(x).get(y).color = total;
31             }
32         }
33         return pixels;
34     }
35 }
```

Abbildung A.3: Beispiel-Datei 3: Convolution.jav

Anhang B

Server-Implementierung

B.1 Interface für serialisierbare Klassen

```
1 public interface ISerializableData {
2
3     public abstract ISerialNode toSerial(KeyStorage keyStorage);
4
5     // Static methods are not inherited in Java.
6     // Meaning this is only a hint to the IDE and programmer
7     // to implement this method
8     public static Object fromSerial(SerialMap data, UnifyContext context) {
9         throw new NotImplementedException(
10             "Missing implementation of \"fromSerial\""
11         );
12     }
13 }
```

Abbildung B.1: Interface: ISerializableData

B.2 Serialisierung einer Java-Klasse

```
1  @Override
2  public SerialMap toSerial(KeyStorage keyStorage) {
3      SerialMap serialized = new SerialMap();
4      serialized.put("ta1", this.TA1.toSerial(keyStorage));
5      serialized.put("ta2", this.TA2.toSerial(keyStorage));
6      serialized.put("op", this.eOperator.toString());
7      serialized.put("noUnification", this.noUnification);
8      serialized.put("location", this.location == null ? null
9          : this.location.toSerial(keyStorage));
10     return serialized;
11 }
12
13 public static Pair fromSerial(SerialMap data, UnifyContext context) {
14     String op = data.getValue("op").getOf(String.class);
15     SerialMap ta1 = data.getMap("ta1");
16     SerialMap ta2 = data.getMap("ta2");
17     boolean noUnification = data.getValue("noUnification").getOf(Boolean.class);
18     SerialMap location = data.getMapOrNull("location");
19
20     var pair = new Pair(
21         RefTypeOrTPHOrWildcardOrGeneric.fromSerial(ta1, context),
22         RefTypeOrTPHOrWildcardOrGeneric.fromSerial(ta2, context),
23         PairOperator.fromString(op),
24         noUnification
25     );
26     if (location != null) pair.location = SourceLoc.fromSerial(location);
27     return pair;
28 }
```

Abbildung B.2: Beispiel: Serialisierung in Pair.java

B.3 Implementierung einer IPacket-Klasse

```
1 public class MessagePacket implements
2     IClientToServerPacket, IServerToClientPacket {
3
4     public String message;
5
6     @JsonIgnore // tells Jackson to ignore a method or property
7     public void onHandle(WebSocket webSocket, SocketClient socketClient) {
8         SocketClient.logger.info("SocketMessage: " + this.message);
9     }
10
11     @JsonIgnore
12     public void onHandle(WebSocket webSocket, SocketServer socketServer) {
13         socketServer.log(this.message, webSocket);
14     }
15 }
```

Abbildung B.3: Packet-Beispiel: MessagePacket

Anhang C

Test-Implementierung

C.1 Paket-Tests zur (De-)Serialisierung

```
1  @Test
2  public void serializeUnifyPair() throws JsonProcessingException {
3      UnifyContext unifyContext = PacketExampleData.createTestContext();
4
5      UnifyPair original = PacketExampleData.getExampleUnifyPair(
6          unifyContext, "de.test."
7      );
8      UnifyPair reconstruction = serializeAndDeserialize(
9          original, unifyContext,
10         (o,k) -> UnifyPair.fromSerial((SerialUUID) o, unifyContext, k)
11     );
12
13     // include all properties in the equality check by checking
14     // the class and string-representation as well, because
15     // the class can have a custom equals method
16     assertEquals(original.getClass(), reconstruction.getClass());
17     assertEquals(original.toString(), reconstruction.toString());
18     assertEquals(original, reconstruction);
19 }
```

Abbildung C.1: Test-Beispiel: Unify-Pair Serialisierung