Masterarbeit

Statische Typsysteme für JavaScript

Entwicklung eines Transpilers von Flow nach TypeScript

Zur Erlangung des akademisches Grades eines
Master of Science

angefertigt von Jonathan Gruber (68341)

Fakultät Informatik, Mathematik und Naturwissenschaften
Studiengang: Master Informatik (16INM/VZ)

Erstprüfer	Prof. Dr. rer. nat. habil. Frank
Zweitprüfer	M. Sc. Michael Lückgen

ausgegeben am 11. Juni 2019
abgegeben am TBA 2019

Zusammenfassung

Die vorliegende Masterarbeit beschäftigt sich mit Transpilierung statischer Typsysteme für JavaScript. Dabei werden zwei derzeit populäre Systeme betrachtet:

Inhaltsverzeichnis

1 Motivation **1**

1.1 JavaScripts Typsystem 1

1.2 Sinnhaftigkeit statischer Typsysteme 2

1.3 Zielsetzung und Aufbau der Arbeit 2

2 Grundlagen **3**

2.1 Statische Typsysteme für JavaScript 3

2.1.1 Flow 3

2.1.2 TypeScript 5

2.2 Compiler und Transpiler 5

2.2.1 Lexikalische Analyse 5

2.2.2 Syntaxanalyse 5

2.2.3 Parser und Transpiler für JavaScript 6

3 Ziel- und Anforderungsanalyse **7**

3.1 Beschreibung der Ausgangslage 7

3.2 Ziele der angestrebten Migration zu TypeScript 7

3.2.1 Erkennung neuer Bugs und Typfehler 8

3.2.2 Unterstützung externer Bibliotheken und Frameworks 8

3.2.3 Stabilität und Geschwindigkeit des Typsystems 8

3.2.4 Zukunftssicherheit und Transparenz der Technologie 8

3.3 Anforderungen an den Transpiler 8

3.3.1 Korrekte Übersetzung der Flow-Typen nach TypeScript 8

3.3.2 Semantisch äquivalente Transpilierung des Quelltexts 8

3.3.3 Verarbeitung gesamter Projektverzeichnisse 8

3.3.4 Beibehaltung der Quelltext-Formatierung 8

4 Umsetzung **9**
4.1 Software-Architektur . . . 10
4.1.1 Funktionsweise von Babel-Plugins . . . 10
4.1.2 Konzeptioneller Aufbau des Transpilers . . . 11
4.2 Entwicklungsprozess . . . 11
4.3 Implementierung als Babel-Plugin . . . 12
4.3.1 Transpilierung der Basistypen . . . 12
4.3.2 Transpilierung der Hilfstypen . . . 12
4.3.3 Transpilierung der Deklarationen . . . 12
4.3.4 Weitere Optimierungen . . . 12
4.4 Erweiterung als Kommandozeilenprogramm . . . 12
4.5 Formatierung des Ausgabequelltexts . . . 13

5 Durchführung **14**
5.1 Transpilierung der Projekte . . . 14
5.2 Manuelle Behebung neuer Typfehler . . . 14

6 Auswertung und Diskussion **15**
6.1 Bewertung der Ergebnisse hinsichtlich der Zielvorgabe . . . 15
6.1.1 Semantische Äquivalenz des Ausgabeprogramms . . . 15
6.1.2 Erkennung neuer Typ- und Programmfehler . . . 15
6.1.3 Verfügbarkeit und Qualität externer Typdefinitionen . . . 15
6.1.4 Performance der Typüberprüfungen mittels TypeScript . . . 15
6.1.5 Formatierung des Ausgabequelltexts . . . 15
6.2 Vergleich des Transpilers mit konkurrierenden Ansätzen . . . 15
6.3 Zusammenfassung . . . 15

7 Schlussbetrachtung **16**
7.1 Zusammenfassung . . . 16
7.2 Ausblick . . . 16

Literatur **I**

Abbildungsverzeichnis **III**

Tabellenverzeichnis **IV**

Quelltextverzeichnis **V**

Eidesstattliche Erklärung **VI**

A Quelltexte **VII**

A.1 Transpiler (Reflow) VII

1 Motivation

1.1 JavaScripts Typsystem

Als JavaScript 1995 von Brendan Eich innerhalb von lediglich zehn Tagen als Bestandteils des Webbrowsers *Netscape Communicator* entworfen wurde [10], war nicht abzusehen, welch enorme Bedeutung die Sprache über 20 Jahre später inne haben wird: Heute wird JavaScript oft als die am weitesten verbreitete Programmiersprache der Welt betrachtet. Dies belegt beispielsweise die alljährliche Umfrage „Stack Overflow Developer Survey“ der Programmierer-Plattform „Stack Overflow“. Diese wertet die Ergebnisse der weltweiten Befragung von circa 90.000 Software-Entwicklern aus [6]. Bereits das siebte Jahr in Folge führt JavaScript dort die Rangliste der populärsten Programmiersprachen an. Der seit vielen Jahren anhaltende Trend, dass zunehmend mehr Software als Webanwendung statt konventioneller Desktop-Anwendung konzipiert wird, hat wesentlich zum Bedeutungszuwachs JavaScripts beigetragen [11][1]. Mit Aufkommen der Laufzeitumgebung „Node.js“ [3] hat sich die Sprache weiterhin von der Lingua Franca des Webbrowsers zu einer ernst zu nehmenden Alternative zu klassischen serverseitigen Programmiersprachen wie Java oder C# entwickelt [12].

Die heutige große Beliebtheit der Sprache steht in starken Kontrast zu ihren Anfängen: Aufgrund der sehr unheitlichen Implementierung in den verschiedenen Webbrowsern und des dynamischen, inkonsistenten Typsystems wurde JavaScript von professionellen Software-Entwicklern zu Beginn als eine mangelhaft entworfene Programmiersprache betrachtet und daher mehrheitlich abgelehnt [2]. Das Typsystem der Skriptsprache ist notorisch bekannt für die impliziten, z. T. unbeabsichtigten, Typumwandlungen von Variablen. Der folgende Quelltext veranschaulicht einige dieser Inkonsistenzen und überraschenden Effekte anhand einiger Beispiele:

```
// Array-Literal plus Array-Literal
[] + []  // -> '' (leerer String)

// Array-Literal plus Objekt-Literal
[] + {}  // -> '[object Object]'

// Erwartung: Plus-Operation sollte kommutativ sein
{} + []  // -> jedoch: 0

// Objekt-Literal plus Objekt-Literal
{} + {}  // -> '[object Object][object Object]'
```

Quelltext 1: Vergleich der zwei Ansätze für statische Typisierung von JavaScript mit Flow (oben) und TypeScript (unten).

1.2 Sinnhaftigkeit statischer Typsysteme

Unbeabsichtigte, implizite Typumwandlungen zur Laufzeit eines Programms und falsche Annahmen über vorliegende Datenstrukturen sind häufige Ursache von Bugs. Eine explizite, statische Typisierung, wie sie beispielsweise in C++ oder Haskell vorliegt ist erstrebenswert, da sie viele Vorteile für den Software-Entwicklungsprozess bietet:
Logik- und Flüchtigkeitsfehler im Quelltext können oftmals bereits vor Ausführung des Programms erkannt und behoben werden. Software-Entwickler gewinnen Sicherheit und Zuversicht, dass Änderungen in umfangreichen Projekten keine unerwünschte Nebenwirkung verursachen, wodurch sich die Wartbarkeit der Software erhöht. Darüber hinaus zwingt die Deklaration expliziter Typen den Programmierer dazu seine *Intension* klar zu formulieren, wodurch sich die Ausdruckskraft des Codes verbessert. Durch eine vernünftige Typisierung wird der Quelltext weiterhin bereits grundlegend dokumentiert („*inline documentation*“).

Überleitung: genau deswegen brauchen wir statische Typsysteme. Da hätten wir zwei…

1.3 Zielsetzung und Aufbau der Arbeit

2 Grundlagen

… und diese werden nun sogleich näher beschrieben:

2.1 Statische Typsysteme für JavaScript

Es gibt noch viele weitere: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

2.1.1 Flow

Flow beschreiben (und zwar mit entsprechender Fachsprache)

Flow-Typ	Beispiel
Array type	`Array<number>`
Boolean literal type	`true`
Boolean type	`boolean`
Empty type	`empty`
Exact object type	`{| prop: any |}`
Function type	`(string, {}) => number`
Generic type annotation	`let v: <FlowType>`
Generics	`type Generic<T: Super> = T`
Interface type	`interface { +prop: number }`
Intersection type	`type Intersection = T1 & T2`
Mixed type	`mixed`
Null literal type	`null`
Nullable type (Maybe type)	`?number`
Number literal type	`42`
Number type	`number`
Object type	`{ [string]: number }`
Opaque type	`opaque type Opaque = number`
String literal type	`'literal'`
String type	`string`
This type	`this`
Tuple type	`[Date, number]`
Type alias	`type Type = <FlowType>`
Type casting	`(variable: string)`
Typeof type	`typeof undefined`
Union type	`number | null`
Void type	`void`

Tabelle 2.1: Basistypen von Flow mit Beispiel

Typ	Beispiel
Type imports	`import type T from './types'`
Type exports	`export type T = number | null`

Tabelle 2.2: Syntax von Typexporten und -importen

Basistypen

Hilfstypen

Deklarationen

Typ-Importe und -Exporte

2.1.2 TypeScript

TS beschreiben (und zwar mit entsprechender Fachsprache)

2.2 Compiler und Transpiler

Was macht eigentlich so ein Compiler bzw. Transpiler? Hier Theorie (AST etc.)

2.2.1 Lexikalische Analyse

Quelltext (string) => Tokens

Parser, Tokenizer = Lexer

2.2.2 Syntaxanalyse

Tokens => AST

2.2.3 Parser und Transpiler für JavaScript

Babel als populärer Vertreter eines JavaScript-Compilers

3 Ziel- und Anforderungsanalyse

3.1 Beschreibung der Ausgangslage

Was geht so bei TeamShirts? Wie verwenden wir Flow? Wie ist die Typisierung bisher (eher implizit, explizit etc)?

Erläutern, dass es da schon eine Handvoll Ansätze auf GitHub gab, aber die alle nicht einsatzbereit waren.

3.2 Ziele der angestrebten Migration zu TypeScript

Hypothesen, Wünsche, Hoffnungen

3.2.1 Erkennung neuer Bugs und Typfehler

3.2.2 Unterstützung externer Bibliotheken und Frameworks

3.2.3 Stabilität und Geschwindigkeit des Typsystems

3.2.4 Zukunftssicherheit und Transparenz der Technologie

3.3 Anforderungen an den Transpiler

3.3.1 Korrekte Übersetzung der Flow-Typen nach TypeScript

3.3.2 Semantisch äquivalente Transpilierung des Quelltexts

3.3.3 Verarbeitung gesamter Projektverzeichnisse

3.3.4 Beibehaltung der Quelltext-Formatierung

4 Umsetzung

Nachdem die Ziele der angestrebten TypeScript-Migration charakterisiert und die Anforderungen an den geplanten Transpiler ausgeführt wurden, soll im Folgenden der Entwurf und die Details der entsprechend gewählten Implementierung ausgeführt werden. In Abschnitt 2.2.3 wurden bereits die verbreitetsten Werkzeuge im Umfeld der Transpilierung von JavaScript ausführlich vorgestellt und verglichen. Auf Basis dieser Gegenüberstellung wurde schließlich Babel [8] als Grundlage der vorliegenden Implementierung des Transpilers von Flow nach TypeScript gewählt.

Im Gegensatz zu den betrachteten Alternativen unterstützt lediglich Babel die Syntax von Flow, TypeScript und moderner bzw. experimenteller JavaScript-Sprachkonstrukte vollständig. Dieser Aspekt ist entscheidend, da nur so eine universelle Übersetzung *jeglicher* Flow-Syntax in äquivalentes TypeScript umgesetzt werden kann[1]. Ein weiteres Argument für die Wahl von Babel ist einerseits die sehr gute Erweiterbarkeit durch ein Plugin-System, andererseits die Ausgereiftheit und große Verbreitung des Projekts. Keine der anderen Optionen konnte die Anforderungen des Transpilers in vergleichbarem Maße erfüllen.

[1] vgl. Anforderung 3.3.1

4.1 Software-Architektur

4.1.1 Funktionsweise von Babel-Plugins

Der Kern von Babel selbst setzt sich aus einer Vielzahl von Plugins zusammen, welche in ihrer Gesamtheit die Funktionalität des Compilers realisieren [8]. Dies verdeutlicht die tiefgreifend integrierte Modularität des Systems. Plugins stellen die elementaren Bausteine dar, welche eine flexible Erweiterung des Compilers ermöglichen. Mit der Entscheidung den Flow-Transpiler als Babel-Plugin umzusetzen, ist dessen Grundarchitektur bereits in Teilen festgelegt, da alle Plugins die vorgegebenen Programmschnittstellen von Babel implementieren müssen. Der konzeptionelle Ablauf eines Plugins gliedert sich in folgende drei Phasen [7]:

1. **Parsen des Eingabecodes**
   Zunächst wird der ursprüngliche Quelltext in zwei Schritten eingelesen, um den abstrakten Syntaxbaum (AST) des Programms zu erzeugen: Als Erstes wird der Code während der lexikalischen Analyse mittels des Tokenizers in Tokens zerlegt. Anschließend werden diese in der syntaktischen Analyse zu einer Datenstruktur umgeformt, die den zugehörigen Syntaxbaum repräsentiert.

2. **Transformation des Programms**
   Während der zweiten Phase wird daraufhin die eigentliche Programmtransformation durchgeführt: Dabei wird der abstrakte Syntaxbaum mittels des *Besucher*-Entwurfsmusters rekursiv traversiert und die Knoten des Baums sukzessive modifiziert, gelöscht bzw. neu erstellte Elemente eingefügt [7]. Das Besucher-Entwurfsmuster[2] beschreibt, wie Operationen auf einer Objektdatenstruktur, unabhängig von der konkreten Implementierung der zugrunde liegenden Klassen, realisiert werden können [4, S. 634 f.]. Im vorliegenden Fall ermöglicht die Anwendung des Musters die gewünschte Menge der Knoten des Syntaxbaums individuell zu „besuchen“ und dort gewünschte Transformation des Programms durchzuführen.

[2] Dieses Entwurfsmuster (*engl. Visitor-Pattern.*) gehört zu den 23 Entwufsmustern, welche im Standardwerk *Design Patterns: Elements of Reusable Object-Oriented Software* der „Gang of Four“ (E. Gamma, R. Helm, R. Johnson und J. Vlissides) beschrieben wird [5, S. 306 ff.].

3. **Generierung des Ausgabequelltexts**
   Schließlich kann der Ausgabecode generiert werden: Hierbei werden alle Knoten des abstrakten Syntaxbaums durch Anwendung einer Tiefensuche nach und nach durchlaufen und eine Zeichenkette aufgebaut, welche dem modifizierten, endgültigen Quelltext entspricht.

Im Folgenden wird das Hauptaugenmerk der Betrachtung auf die zweite Phase gelegt, da dort die vorliegende Problemstellung der Transpilierung von Flow- nach TypeScript-Code gelöst wird. Das Parsen der Eingabe und das Generieren der Ausgabe kann simpel durch Verwendung der gegebenen Standard-Funktionen von Babel umgesetzt werden.

4.1.2 Konzeptioneller Aufbau des Transpilers

Abbildung <TODO> zeigt den konzeptionellen Aufbau der umgesetzten Implementierung als Babel-Plugin.

4.2 Entwicklungsprozess

Irgendwo sollte wohl geschrieben werden, dass TypeScript, TDD usw. verwendet wurde, um das Plugin zu bauen…

4.3 Implementierung als Babel-Plugin

4.3.1 Transpilierung der Basistypen

4.3.2 Transpilierung der Hilfstypen

4.3.3 Transpilierung der Deklarationen

4.3.4 Weitere Optimierungen

Übersetzung gängiger Typimporte

Konvertierung von Class Decorators

Mapping der Importe (verschiedene Typnamen in Flow und TS), Umwandlung der Decorators usw.

4.4 Erweiterung als Kommandozeilenprogramm

Aufgrund der in Abschnitt 3.3.3 dargelegten Anforderung, dass der Transpiler in der Lage sein muss gesamte Projektverzeichnisse verarbeiten zu können, ist eine Erweitung als Kommandozeilenprogramm naheliegend. Dieses stellt lediglich eine schmale Ummantelung des Babel-Plugins dar

4.5 Formatierung des Ausgabequelltexts

Prettier, synchronisieren der Leerzeilen und Kommentare beschreiben usw.

5 Durchführung

5.1 Transpilierung der Projekte

5.2 Manuelle Behebung neuer Typfehler

6 Auswertung und Diskussion

6.1 Bewertung der Ergebnisse hinsichtlich der Zielvorgabe

6.1.1 Semantische Äquivalenz des Ausgabeprogramms

6.1.2 Erkennung neuer Typ- und Programmfehler

6.1.3 Verfügbarkeit und Qualität externer Typdefinitionen

6.1.4 Performance der Typüberprüfungen mittels TypeScript

6.1.5 Formatierung des Ausgabequelltexts

6.2 Vergleich des Transpilers mit konkurrierenden Ansätzen

6.3 Zusammenfassung

7 Schlussbetrachtung

7.1 Zusammenfassung

7.2 Ausblick

Literatur

[1] Sven Casteleyn, Irene Garrig'os und Jose-Norberto Maz'on. „Ten years of rich internet applications: A systematic mapping study, and beyond“. In: *ACM Transactions on the Web (TWEB)* 8.3 (2014), S. 18 (siehe S. 1).

[4] Elisabeth Freeman u. a. *Head First Design Patterns*. 2nd. O'Reilly & Associates, Inc., 2004. ISBN: 978-0-5960-07126 (siehe S. 10).

[5] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Aufl. Addison-Wesley Professional, 1994. ISBN: 0201633612 (siehe S. 10).

[7] Jamie Kyle, Sebastian McKenzie, Henry Zhu u. a. *Babel Handbook*. Babel. Sep. 2017. URL: https://github.com/jamiebuilds/babel-handbook/blob/master/translations/en/user-handbook.md (besucht am 11. 07. 2019) (siehe S. 10).

[9] *MIT License*. Massachusetts Institute of Technology. URL: https://opensource.org/licenses/MIT (besucht am 01. 07. 2019) (siehe S. VII).

[10] Charles Severance. „JavaScript: Designing a Language in 10 Days“. In: *Computer* 45 (Feb. 2012), S. 7–8. ISSN: 0018-9162. DOI: 10.1109/MC.2012.57. URL: doi.ieeecomputersociety.org/10.1109/MC.2012.57 (siehe S. 1).

[11] Antero Taivalsaari und Tommi Mikkonen. „The Web as a Software Platform: Ten Years Later“. In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,* INSTICC. SciTePress, 2017, S. 41–50. ISBN: 978-989-758-246-2. DOI: 10.5220/0006234800410050 (siehe S. 1).

[12] S. Tilkov und S. Vinoski. „Node.js: Using JavaScript to Build High-Performance Network Programs“. In: *IEEE Internet Computing* 14.6 (Nov. 2010), S. 80–83. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145 (siehe S. 1).

Online-Quellen

[2] Steve Champeon. *JavaScript: How Did We Get Here?* Juni 2001. URL: https://web.archive.org/web/20160719020828/http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html (besucht am 25. 03. 2019) (siehe S. 1).

[6] Stack Exchange Inc. *Stack Overflow Annual Developer Survey*. 2019. URL: https://insights.stackoverflow.com/survey/2019 (besucht am 03. 06. 2019) (siehe S. 1).

Software

[3] Node.js Foundation. *Node.js*. 2019. URL: https://nodejs.org (besucht am 04. 06. 2019) (siehe S. 1).

[8] Sebastian McKenzie und other contributors. *Babel - The compiler for next generation JavaScript*. 2018. URL: https://babeljs.io/ (besucht am 22. 03. 2019) (siehe S. 9 f.).

Abbildungsverzeichnis

Tabellenverzeichnis

2.1 Basistypen von Flow mit Beispiel 4
2.2 Syntax von Typexporten und -importen 4

Quelltextverzeichnis

1 Vergleich der zwei Ansätze für statische Typisierung von JavaScript mit Flow (oben) und TypeScript (unten). 2

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst habe. Ich versichere, dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahren gewesen ist.

Leipzig, den 14. Juli 2019

_____

Jonathan Gruber

A Quelltexte

A.1 Transpiler (Reflow)

Der Quelltext des im Zuge dieser Arbeit entwickelten Transpilers ist unter folgendem GitHub-Repository vollständig einsehbar:

https://github.com/grubersjoe/reflow

Das Projekt wurde unter der MIT License [9] veröffentlicht.