HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Masterarbeit

Statische Typsysteme für JavaScript

Entwicklung eines Transpilers zur Übersetzung von Flow nach TypeScript

Zur Erlangung des akademisches Grades eines
Master of Science

angefertigt von Jonathan Gruber (68341)

Fakultät Informatik und Medien
Studiengang: Master Informatik (16INM/VZ)

Erstprüfer	Prof. Dr. rer. nat. habil. Michael Frank
Zweitprüfer	M. Sc. Michael Lückgen (sprd.net AG)

Leipzig, den 4. Dezember 2019

Zusammenfassung

Die vorliegende Masterarbeit behandelt die Systeme Flow [24] und TypeScript [125], die eine statische Typisierung in JavaScript ermöglichen. Dabei wird die Implementierung eines Transpilers auf Basis von Babel [93] ausgeführt, der beliebige durch Flow typisierte JavaScript-Programme in äquivalenten TypeScript-Code übersetzen kann. Mittels des entwickelten Transpilers wurden zwei reale Projekte des Unternehmens Spreadshirt erfolgreich nach TypeScript migriert. Es kann auf Grundlage empirischer Daten, die anhand der übersetzten Projekte gewonnen wurden, gezeigt werden, dass TypeScript verschiedene Vorteile im Vergleich zu Flow aufweist: So sind durch den Wechsel des Typsystems neue Programmfehler aufgedeckt worden, und es kann belegt werden, dass TypeScript eine umfangreichere Unterstützung für externe Softwarebibliotheken mittels Typdefinitionen bietet. Darüber hinaus geht aus der Untersuchung hervor, dass TypeScript transparenter als Flow entwickelt wird. Schließlich kann in einigen Fällen bei TypeScript ein besseres Laufzeitverhalten der Typüberprüfungen gegenüber Flow nachgewiesen werden.

Inhaltsverzeichnis

1 Motivation

1.1 Historischer Abriss von JavaScript

Als JavaScript 1995 von Brendan Eich in nur zehn Tagen als Bestandteil des Webbrowsers *Netscape Communicator* entworfen wurde [113], war nicht abzusehen, welche Relevanz die Sprache über 20 Jahre später besitzen würde: Heute wird JavaScript oft als die am weitesten verbreitete Programmiersprache der Welt angesehen [104, 32]. Dies belegt beispielsweise die alljährliche Umfrage „*Stack Overflow Developer Survey*“ der Programmierer-Plattform *Stack Overflow*, welche die Ergebnisse der weltweiten Befragung von circa 90.000 Softwareentwicklern auswertet [79]. 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 realisiert wird, hat die Bedeutung von JavaScript stark erhöht [121, 21].

Die heutige große Beliebtheit der Sprache steht in starkem Kontrast zu ihren Anfängen. Zunächst besaß JavaScript eine zweifelhafte Reputation und wurde aus verschiedenen Gründen von professionellen Softwareentwicklern als eine mangelhaft entworfene Programmiersprache angesehen [32]: Die uneinheitliche und teilweise fehlerbehaftete Implementierung in den frühen Webbrowsern und der Mangel an Entwicklungswerkzeugen wie Debuggern erschwerte die Programmierung von JavaScript-Anwendungen enorm [22]. Darüber hinaus behinderte die anfängliche Unbeständigkeit der Sprache und die „schlechte Qualität“ [31] der ersten Versionen der ECMAScript-Spezifikation[1] [43] eine breite Akzeptanz der Sprache in der damaligen Entwickler-Szene. Ein weiterer Aspekt von JavaScript, der bis heute kritisch betrachtet wird, ist die dynamische Typisierung der Sprache, weil diese die Entwicklung sicherer und korrekter Software erschwert [120, 105]. Im Folgenden soll dieser Gesichtspunkt näher beleuchtet werden, um die Problematik zu verdeutlichen.

[1] „ECMAScript“ ist der Name der formalen Spezifikation von JavaScript und geht auf die Normungsorganisation *Ecma International* zurück. *Netscape Communications* beauftragte diese im November 1996 mit der Erstellung des Standards, welcher im Jahr darauf unter der Bezeichnung „ECMA-262“ veröffentlicht wurde [43].

1.2 JavaScripts dynamische Typisierung

Eine dynamische Typisierung ist dadurch charakterisiert, dass jedem Wert der Programmiersprache zur *Laufzeit* ein Typ zugewiesen wird und sich dieser je nach Striktheit des Typsystems zu einem späteren Zeitpunkt implizit oder explizit ändern kann [126, S. 45]. Typfehler werden dabei durch dynamische Überprüfungen der Laufzeitumgebung erkannt [20, S. 37]. JavaScript besitzt die folgenden sieben Datentypen [42, S. 25]. Diese legen fest, wie Werte des Programms zur Laufzeit interpretiert werden, welchen Wertebereich sie umfassen und welche Operationen mit ihnen möglich sind. Alle Datentypen außer `Object` sind dabei primitiv, das heißt nicht weiter zerlegbar (atomar) [42, S. 8].

- `Undefined`: Undefinierte Werte
- `Null`: Nullwert
- `Boolean`: Boolesche Werte
- `String`: Zeichenketten
- `Symbol`: Eindeutige Bezeichner
- `Number`: Gleitkommazahlen
- `Object`: Alles Weitere (Felder, Funktionen, Objekte, usw.)

Der Typ von Werten wird in JavaScript während der Programmausführung bei Bedarf automatisch implizit umgewandelt (*type coercion*) [106]. Einerseits kann argumentiert werden, dass eine derartige Flexibilität den Entwicklungsprozess beschleunigt [24, S. 1], andererseits birgt dies das Risiko von Programmfehlern, da Typverletzungen erst zur Laufzeit festgestellt werden können [8]. Weil eine Typisierung von Ausdrücken syntaktisch nicht möglich ist, ist nicht immer offensichtlich, welche Datentypen in JavaScript-Quelltexten vorliegen. Deshalb ist die Programmierung so fehleranfällig. Problematisch für die Entwicklung korrekter Anwendungen ist darüber hinaus, dass JavaScript viele Operationen, die traditionell als dynamischer Typfehler gelten, nicht als solche behandelt, sondern die Programmausführung fortsetzt [120]. So kann beispielsweise auf nicht-existente Attribute von Objekten zugegriffen werden, ohne dass dies eine Typverletzung verursacht. Infolgedessen ist es möglich, dass inkorrektes Programmverhalten zunächst unentdeckt bleibt, weil nicht unmittelbar ein fataler Fehler auftritt.

1.3 Motivation für den Einsatz statischer Typsysteme

Viele der anfänglichen Unzulänglichkeiten von JavaScript konnten durch Erweiterungen der Spezifikation und zunehmend konsistenter Implementierungen nach und nach behoben werden. Swamy et al. führen aber aus, dass die dynamische Typisierung der Sprache weiterhin ein Hindernis für die Entwicklung sicherer und korrekter Anwendungen darstellen kann [120]. Auch Bierman et al. vertreten die These, dass JavaScript nach wie vor ungeeignet sei für die Entwicklung und Wartung umfangreicher Software [17, S. 1]. Über die Jahre sind verschiedene Ansätze entstanden, um diese Problematik zu überwinden, indem JavaScript um ein *statisches* Typsystem erweitert wird. Der wesentliche Zweck statischer Typsysteme ist die Vermeidung von Programmfehlern [20, S. 1]. Dies wird durch die statische Analyse des Quelltexts realisiert, sodass Typverletzungen bereits vor der Programmausführung festgestellt werden können. Im Gegensatz zu einer dynamischen Typisierung wird der Typ von Ausdrücken in statischen Systemen explizit deklariert oder kann selbstständig inferiert (abgeleitet) werden [126, S. 45]. Eine *explizite* Typisierung bedeutet, dass Typdeklarationen Bestandteil der Syntax der Programmiersprache bzw. des Systems sind [20, S. 2].

Ein statisches Typsystem bietet verschiedene Vorteile für den Softwareentwicklungsprozess: Da Typen als ein Attribut angesehen werden können, die eine bestimmte Eigenschaft des Programms mittels der Mechanismen des Typsystems beweisen, tragen sie zur Verifizierung der Programmkorrektheit bei [115]. Da der Compiler statisch berechnen kann, ob bei Ausdrücken und Anweisungen Typverletzungen auftreten, können semantische Probleme wie Logik- und Flüchtigkeitsfehler aufgedeckt und korrigiert werden. Sofern alle Schnittstellen und Datenstrukturen typisiert sind, wird deren inkorrekte Verwendung unmittelbar erkannt [20, S. 6]. Weiterhin ist es möglich, die Einschränkungen und Regeln der gegebenen Anwendungsdomäne durch des Typsystems präzise zu modellieren und deren Einhaltung mit Hilfe des System zu erzwingen. Auch erfordert die Deklaration expliziter Typen, dass der Entwickler die Absicht seines Programms klar formuliert, wodurch sich die Ausdruckskraft und Lesbarkeit des Codes erhöht [126, S. 96]. Durch eine Typisierung wird das Verhalten und die Struktur der Software darüber hinaus bereits grundlegend dokumentiert [98, Abschn. 6.1.1].

Derzeit existieren zwei populäre Technologien, die eine statische Typisierung in JavaScript ermöglichen und so die Defizite des dynamischen Typsystems ausgleichen: einerseits *Flow* [24], andererseits *TypeScript* [125]. Eine nach wie vor ungelöste Problemstellung ist jedoch die

äquivalente Übersetzung dieser Systeme ineinander. Die Motivation für eine derartige Transformation ist es, den Wechsel des eingesetzten statischen Typsystems mit möglichst geringem manuellen Aufwand zu realisieren. Die händische Übertragung umfangreicher Projekte ist impraktikabel, weil dies sehr zeitaufwändig und fehleranfällig wäre.

1.4 Zielsetzung und Aufbau der Arbeit

Die vorliegende Masterarbeit beschäftigt sich mit der Lösung dieser aufgeworfenen Problemstellung, der automatischen Migration von Flow nach TypeScript, und ist in Zusammenarbeit mit dem Unternehmen *sprd.net AG* (*Spreadshirt*) entstanden. Ziel der Arbeit ist die Entwicklung eines *Transpilers* (auch Transcompiler genannt), der es ermöglicht, den gesamten Quelltext eines durch Flow typisierten JavaScript-Projekts in äquivalenten TypeScript-Code zu übersetzen. Unter einem Transpiler wird ein spezieller Compiler verstanden, der den Quelltext einer Programmiersprache in eine andere Programmiersprache mit ähnlichem Abstraktionsniveau bedeutungsgleich übersetzt [78]. Der Wechsel des eingesetzten Typsystems wird durch Spreadshirt angestrebt, da angenommen wird, dass TypeScript verschiedener Vorteile gegenüber Flow aufweist. So wird vermutet, dass TypeScript Typ- und Programmfehler in höherem Maße erkennt und externe Softwarebibliotheken besser unterstützt. Außerdem erwartet man sich, dass TypeScript Flow hinsichtlich der Performance, Transparenz und Zukunftssicherheit überlegen ist. Die Verifizierung dieser Thesen auf Grundlage empirischer Daten und der gesammelten Erfahrung während der Projektmigration ist Gegenstand der anschließenden Untersuchung.

Der Aufbau der Arbeit gliedert sich in sechs Kapitel: Nachfolgend werden zunächst die benötigten theoretischen Grundlagen bezüglich statischer Typsysteme und der Transpilierung von Programmen geschaffen. Weiterhin werden die Typsysteme sowohl von Flow, als auch von TypeScript charakterisiert und voneinander abgegrenzt. Anschließend werden die Rahmenbedingungen zu Beginn der Arbeit beschrieben, die Ziele des Wechsels von Flow zu TypeScript erläutert und die Anforderungen an den angestrebten Transpiler ausgeführt. Daraufhin werden Architektur und Details der Implementierung des Übersetzers ausführlich betrachtet. In Kapitel 5 wird die Durchführung der Migration der JavaScript-Projekte von Spreadshirt mittels des umgesetzten Transcompilers dargelegt und gewonnene Erfahrungen beschrieben. Außerdem werden hier die Ergebnisse der Migration ausgewertet, kritisch diskutiert und hinsichtlich der Zielvorgabe bewertet. Bereits heute bestehen Ansätze zur Transpilierung von Flow, die

aber weder vollständig noch fehlerfrei sind. Der realisierte Transpiler wird diesen bestehenden Werkzeugen gegenübergestellt, sodass die erzielten Ergebnisse besser eingeordnet werden können. Zuletzt wird ein Fazit der gesamten Arbeit gezogen und ein Ausblick über mögliche Erweiterungen gegeben.

2 Grundlagen

Bevor die Ziele der Migration von Flow zu TypeScript im nächsten Kapitel detailliert dargelegt werden, sollen zunächst die nötigen theoretischen Grundlagen dargestellt werden, um die Nachvollziehbarkeit der weiteren Ausführungen zu erleichtern. Dazu werden im Folgenden drei Konzepte aus der Theorie der Typensysteme erläutert, die relevant für die Differenzierung von Flow gegenüber TypeScript sind.

2.1 Konzepte und Begriffe der Typentheorie

2.1.1 Korrektheit von Typsystemen

Ein wichtige Eigenschaft von Typsystemen ist deren logische *Korrektheit*[2]. Dieses Kriterium beschreibt, ob das System garantieren kann, dass ein Programm während dessen Ausführung tatsächlich keine Typfehler verursacht, sofern keine statischen Typverletzungen bestehen [129]. In einem solchen System stimmt also der Typ eines zur Laufzeit ausgewerteten Ausdrucks stets mit dem statischen Typ überein [131]. Durch mathematische Formalisierung kann diese Eigenschaft bewiesen werden [20, S. 7]. Das Gegenstück von Korrektheit ist *Vollständigkeit*. Während ein korrektes System alle Fehler identifiziert, die zur Laufzeit auftreten können, findet ein vollständiges System nur diejenigen Fehler, die tatsächlich während der Ausführung eintreten [61]. Im ersten Fall werden unter Umständen Fehler entdeckt, die zur Laufzeit praktisch nicht vorkommen (falsch positiver Fehler), im zweiten Fall treten dagegen eventuell Laufzeitfehler auf, obwohl keine Typverletzung festgestellt wurde (falsch negativer Fehler). Im Idealfall ist ein Typsystem sowohl korrekt als auch vollständig.

Das Typsystem mancher Programmiersprachen ist nicht korrekt. So ist die Semantik bestimmter Operationen in C wie beispielsweise die Dereferenzierung des Nullzeigers in der Sprachspezikation undefiniert [81, S. 79]. Obwohl ein solches Programm durch den Compiler akzeptiert wird, also keine Typverletzungen aufweist, können hier Laufzeitfehler auftreten.

[2] Engl. *soundness*.

2.1.2 Subtyping

Die zwei in dieser Arbeit betrachteten Typsysteme Flow [24] und TypeScript [125] setzen *Subtyping* ein, um die Beziehung zwischen Super- und Subtypen abzubilden [24, 17]. Grundsätzlich kann ein Typ als eine Menge von Werten betrachtet werden, die eine gemeinsame Struktur oder Form besitzen [7, S. 3]. Subtyping stellt die reflexive und transitive Relation dar, die aussagt, ob ein Typ T Untertyp eines anderen Typen S ist [20, S. 27 f.]. Hierfür muss das Kriterium der *Subsumtion* erfüllt sein. Diese besteht, wenn die Menge, die S darstellt, jeden Wert von T beinhaltet. Folglich kann der Typ T überall dort eingesetzt werden, wo S erwartet wird (*Liskovsches Substitutionsprinzip*). Existiert zum Beispiel ein Typ Z, der allen ganzen Zahlen entspricht, und weiterhin ein Typ N, der alle natürlichen Zahlen beinhaltet, so ist N Subtyp von Z, weil Z die Menge N umfasst.

Zwei weitere Begriff in diesem Zusammenhang, die in den weiteren Ausführungen verwendet werden, sind *Kovarianz* und *Kontravarianz*. Dabei bedeutet Kovarianz eine Betrachtung *in* Richtung der Subtyp-Hierarchie, Kontravarianz *entgegen* dieser [19]. Im Fall von Kovarianz werden Subtypen, aber keine Supertypen akzeptiert, bei Kontravarianz hingegen nur Supertypen, aber keine Subtypen [60].

2.1.3 Nominale und strukturelle Typen

Eine Möglichkeit, Typsysteme zu klassifizieren, ist die Verwendung von *nominalen* bzw. *strukturellen* Typen. Da sich Flow und TypeScript in diesem Aspekt unterscheiden, soll auch diese Differenzierung dargelegt werden. Entscheidend ist hierbei, ob unabhängige Typen durch das Typsystem als äquivalent angesehen werden oder nicht [20, S. 9]. Auch die Subtyp-Relation wird hierdurch beeinflusst: Bei strukturellen Typen liegt ein Subtyp vor, wenn dessen Attribute eine Obermenge des anderen Typs sind [91]. Liegt hingegen ein nominaler Typ vor, so besteht eine solche Beziehung nur genau dann, wenn sie explizit syntaktisch deklariert wird[3]. Die Typsysteme der meisten Programmiersprachen setzen sowohl nominale als auch strukturelle Typen ein [20, S. 9]. Anhand eines Beispiels in Pseudocode (Quelltext 2.1) soll der Unterschied verdeutlicht werden.

[3] Zum Beispiel in Java durch das Schlüsselwort „`extends`“.

```
1 class A { prop: string; }
2 class B { prop: string; }
3 function f(arg: A) {}
4 f(new B()); // << Typfehler?
```

Quelltext 2.1: Beispiel zur Differenzierung von nominalen und strukturellen Typen.

In den ersten beiden Zeilen werden zunächst zwei Klassen `A` und `B` definiert. Weiterhin wird eine Funktion `f` angegeben, die einen Parameter vom Typ `A` erwartet. Bei Aufruf dieser Funktion mit einer Instanz der Klasse `B` sind nun zwei Fälle möglich: Entweder spezifiziert das System, dass Typen mit unterschiedlichen Namen stets inkompatibel sind, oder die Typen `A` und `B` werden als äquivalent betrachtet, da ihre Struktur übereinstimmt. Im ersten Fall würde in Zeile 4 eine Typverletzung auftreten, weil der Typ des Ausdrucks `new B()` nicht `A`, sondern `B` ist. Läge hingegen eine strukturelle Typisierung vor, so würde das Programm akzeptiert werden, da der Aufbau der Klassen `A` und `B` identisch ist. Beide enthalten genau ein Attribut `prop` mit demselben Typ `string`.

2.2 Statische Typsysteme für JavaScript

In den nachfolgenden Abschnitten sollen nun die charakteristischen Merkmale der zwei in dieser Arbeit behandelten statischen Typsysteme *Flow* [24] und *TypeScript* [125] näher beleuchtet werden, um deren Unterschiede und Gemeinsamkeiten herauszuarbeiten.

2.2.1 Flow

Charakterisierung

Flow [24] ist ein durch das US-amerikanische Unternehmen *Facebook Inc.* entwickeltes Software-System, das statische Typüberprüfungen in JavaScript durch das Einfügen von Typannotationen in den Quelltext ermöglicht. Derartige Annotationen sind eine syntaktische Erweiterung, die vor Auslieferung des Codes mittels eines geeigneten Werkzeugs wie beispielsweise Babel [93] entfernt werden muss, damit wieder regulärer JavaScript-Code entsteht [51]. Flow ist seit 2014 frei und quelloffen unter der MIT-Lizenz verfügbar [97, 48]. Das Typsystem von Flow wird in

der akademischen Arbeit „*Fast and Precise Type Checking for JavaScript*“ [24] von Chaudhuri et al. ausführlich beschrieben. Die nachfolgenden Erläuterungen beziehen sich vorrangig auf den Inhalt dieser Veröffentlichung.

Flow verfolgt zwei primäre Ziele [59]: Erstens soll eine möglichst hohe Präzision durch die Typüberprüfungen erreicht werden, um möglichst zuverlässige Ergebnisse, das heißt eine geringe Quote falsch positiver und falsch negativer Fehler, zu erzielen. Zweitens müssen diese Ergebnisse selbst bei einer sehr umfangreichen Codebasis schnell berechnet werden können, damit der Workflow des Programmierers nicht verlangsamt wird.

Das erste Ziel wird zum einen anhand einer pfadsensitiven Datenflussanalyse, zum anderen durch die Korrektheit des Typsystems realisiert [59]. Mittels einer solchen Datenflussanalyse kann das Laufzeitverhalten von Software präzise modelliert und so der Typ von Ausdrücken durch Einbeziehung der Programmverzweigungen (der Pfade) auf speziellere Untertypen abgebildet werden (*type refinement*) [128, 24, S. 2]. Um die Korrektheit des Typsystems sicherzustellen, berücksichtigt Flow während der Typüberprüfung von Ausdrücken alle theoretisch möglichen Fälle [61]. Wie ausgeführt birgt diese Präferenz von Korrektheit über Vollständigkeit den Nachteil, dass unter Umständen Typfehler angezeigt werden, die zur Laufzeit tatsächlich gar nicht auftreten. Gleichzeitig steigt aber durch diesen Ansatz die Sicherheit, weil damit die Wahrscheinlichkeit sinkt, dass Laufzeitfehler durch das Typsystem unentdeckt bleiben.

Die zweite Zielvorgabe, die Steigerung der Geschwindigkeit der Typüberprüfungen, wird durch eine Modularisierung des Verfahrens erreicht, sodass die Berechnung anhand mehrerer Threads stark parallelisiert werden kann (*Multithreading*) [24, S. 4]. Flow nutzt den Umstand, dass in modernen JavaScript-Projekten üblicherweise genau eine Datei pro Modul vorliegt. Unabhängige Module können somit auf verschiedenen Prozessorkernen gleichzeitig überprüft werden. Daraufhin werden die Ergebnisse der einzelnen Berechnungen durch einen MapReduce-Algorithmus, der auf einer geteilten Heap-Datenstruktur arbeitet, zum Gesamtergebnis der Typüberprüfung rekombiniert [24, S. 22 f.].

Flow ist in der Lage Typen global zu inferieren [24, S. 25]. Das heißt, der Typ vieler Ausdrücke muss nicht explizit angegeben werden, sondern kann durch das Typsystem selbstständig abgeleitet werden. Deshalb kann bereits mit wenigen expliziten Annotationen eine hohe Abdeckung des gesamten Quelltexts durch das Typsystem erzielt werden. So führen Chaudhuri et al. aus, dass in einer circa 13 Millionen Zeilen umfassenden Codebasis von Facebook im

Median nur 29% aller Stellen, an denen Typannotationen möglich sind, nötig waren, um eine vollständige Abdeckung durch Flow zu erzielen [24, S. 24].

Eine weitere Eigenschaft von Typsystemen ist deren Verwendung von nominalen und strukturellen Typen. Flow behandelt Klassen und opake Typen[4] hierbei nominal, alle anderen Typen dagegen strukturell [56]. Somit werden diese Typen stets als inkompatibel angesehen, selbst wenn ihr struktureller Aufbau dies zuließe.

Architektur

Die Architektur von Flow gliedert sich in einen Server, einen Client und einen Dateisystemüberwacher [24, S. 22]. Der Server liest zunächst den Quelltext des gesamten Projekts ein und überprüft diesen hinsichtlich Typverletzungen. Daraufhin läuft der Prozess im Hintergrund weiter und ist bereit, Anfragen des Clients entgegen zu nehmen. Der Client – zum Beispiel die integrierte Entwicklungsumgebung – kann nun durch Kommandos einerseits den generellen Status der Typüberprüfung abrufen, andererseits spezifischere Informationen wie beispielsweise den Typ eines bestimmten Ausdrucks abfragen. Sobald eine Datei editiert, neu erstellt oder gelöscht wird, wird dies dem Server durch den Dateisystemüberwacher mitgeteilt. Daraufhin überprüft der Server die Typkorrektheit inkrementell auf Grundlage des modifizierten Quelltexts erneut. Dabei werden nur geänderte Module und deren Abhängigkeiten betrachtet, sodass der Berechnungs- und damit Zeitaufwand stark reduziert werden kann. Weil die aktuellen Ergebnisse stets im Arbeitsspeicher vorgehalten werden, können nachfolgende Anfragen des Clients schnell beantwortet werden.

Beispiel

Zur Veranschaulichung, wie Flow konkret benutzt wird, soll in Quelltext 2.2 die Typisierung eines einfachen JavaScript-Programms anhand des Algorithmus der linearen Suche gezeigt werden. In diesem und allen weiteren Quelltexten werden dabei die Schlüsselworte von JavaScript bzw. TypeScript **`fett`** und Typannotationen *`kursiv`* abgedruckt.

Das Programm beginnt in Zeile 1 mit einem speziellen Zeilenkommentar (`@flow`). Durch diese Direktive werden diejenigen Dateien markiert, die von Flow auf Typverletzungen überwacht

[4] Siehe Tabelle 2.1.

```
1  // @flow
2  function linearSearch<T>(list: Array<T>, searchValue: T): number | empty {
3    for (const [index, value] of list.entries()) {
4      if (value === searchValue) {
5        return index;
6      }
7    }
8    throw new Error('Not found');
9  }
10
11 linearSearch<number>([3, 1, 10, 56], 10);        // 2
12 linearSearch<number>([3, 1, 10, 56], 12);        // Exception (Not Found)
13 linearSearch<string>(['foo', 'bar', 'baz'], 3);  // Typfehler (3 ist kein String)
```

Quelltext 2.2: Benutzung von Flow anhand des Algorithmus der linearen Suche.

werden sollen. In Zeile 2 wird anschließend die Funktion `linearSearch` definiert, die den Suchalgorithmus implementiert. Deren formale Parameter werden durch einen generischen Typparameter `T`, der in spitzen Klammern hinter dem Funktionsnamen deklariert wird, typisiert. Typannotationen werden durch Angabe eines Doppelpunkts, gefolgt von einem Typ notiert. Auf diese Weise wird festgelegt, dass der erste Parameter `list` ein homogenes Feld mit Werten des Typs `T` und der zweite Parameter `searchValue` ein Wert desselben Typs sein muss. Mit Hilfe dieser Einschränkung werden unsinnige Funktionsaufrufe mit Suchwerten, die aufgrund der Typisierung gar nicht Bestandteil des Felds sein *können*, statisch erkannt (vgl. Zeile 13). Dabei wird der Typparameter `T` beim Aufruf der Methode explizit gesetzt. Der Algorithmus liefert entweder den Feldindex des Suchwerts zurück, sofern dieser in der Liste vorkommt, oder löst eine *Exception* aus[5]. Flow bietet für den zweiten Fall den adäquaten Typ `empty`, welcher dem leeren Typ $\bot$ (*bottom type*) entspricht. Der korrekte Rückgabetyp der Funktion ist somit der Vereinigungstyp aus `number` und `empty`. Dieser wird durch einen senkrechten Strich notiert und hinter der Auflistung der formalen Parameter deklariert. Die Typen der Ausdrücke innerhalb des Funktionskörpers müssen nicht explizit annotiert werden, da Flow diese inferieren kann.

Typen

Nachfolgend sollen nun alle Sprachkonstrukte von Flow kurz beschrieben werden, um das Verständnis der späteren Erläuterung der Transpilierung zu vereinfachen. Die Syntax lässt

[5] Es ist kritisch anzumerken, dass das Nichtauffinden eines Werts im eigentliche Sinne keine Ausnahme (*Exception*) darstellt und hier nur zur Verbesserung der Ausdruckskraft des Beispiels dient.

sich in drei Kategorien einordnen: Basistypen, Hilfstypen und Typdeklarationen. Unter Basistypen werden in dieser Arbeit die regulären Typannotationen von Flow verstanden. Diese machen den größten Teil der Online-Dokumentation [58] aus. Hilfstypen (*Utility types*) sind spezielle Typen, die einen oder mehrere andere Typen als Argument erhalten und so einen neuen Typ mit zusätzlichen, nützlichen Eigenschaften berechnen. Typdeklarationen ermöglichen es schließlich, die Schnittstellen externer Bibliotheken und Frameworks durch spezielle Deklarationsdateien zu definieren. Auf diese Weise kann auch die Benutzung untypisierter Abhängigkeiten statisch überprüft werden.

Basistypen

Tabelle 2.1 listet die Basistypen von Flow auf, beschreibt deren Zweck und zeigt ein Beispiel. Um die Nachvollziehbarkeit zur Online-Dokumentation und der in Kapitel 4 ausgeführten Implementierung zu erleichtern, werden die englischen Typbezeichnungen beibehalten und nicht ins Deutsche übersetzt. Die Beispiele und Erläuterungen versuchen den Großteil der Funktionalität von Flow zu veranschaulichen, jedoch können nicht alle Details ausführlich behandelt werden, da dies den Umfang der Arbeit überschreiten würde.

Tabelle 2.1: Basistypen von Flow [58] mit Beispiel.

Basistyp	**Beispiel**	**Kurzbeschreibung**
Any type	`any`	Beliebige Werte. Jeder Typ ist Subtyp von *any*. *any* ist jedem Typ zuweisbar und umgekehrt.
Array type	`Array<number>` `number[]`	Felder. Der Typparameter (hier `number`) gibt den Typ der Feldelemente an. Die zweite Notation ist eine äquivalente Kurzschreibweise.
Boolean literal type	`true`	Boolesche Literale (entweder `true` oder `false`).
Boolean type	`boolean`	Boolesche Werte.
Class type	`class C {}` `type ClassAlias = C`	Typ für Klassen (durch Klassennamen).
Empty type	`empty`	Der leere Typ $\perp$ (*bottom type*), also der Typ mit genau null Elementen. Nützlich, um niemals terminierende Unterprogramme zu typisieren (zum Beispiel Endlosschleife oder Exception).
Exact object type	`{| prop: boolean; |}`	Objekte mit *genau* der angegebenen Menge von Attributen. Weitere Attribute stellen eine Typverletzung dar (vgl. *Object type*).

Basistypen von Flow [58] mit Beispiel.

Basistyp	**Beispiel**	**Kurzbeschreibung**
Function type	`(string, arg?: {}) => number`	Funktionen: Das heißt, der Typ der formalen Parameter und des Rückgabewerts. Die Parameternamen sind dabei optional.
Generic type annotation	`let v: <FlowType>`	Allgemeine Typannotation für Ausdrücke wie die Deklaration von Variablen, Funktionsparameter, -rückgabewerte et cetera.
Generics	`type Generic<T: Super> = T`	Generische Typen (*parametrische Polymorphie*). `T` ist hierbei Typparameter, `Super` ein zugehöriger Supertyp, der mögliche Werte für T einschränkt.

Interface type	`interface I {`
	`  prop: mixed;`
	`  +covariant: number[];`
	`}`
		Schnittstellen. Wie bei Objekttypen kann ein Attribute durch `+` als kovariant (nur lesbar) oder durch `-` als kontravariant (nur schreibar) markiert werden.

Intersection type	`type Intersection = A & B`	Schnittmenge zweier Typen. Der Typ `Intersection` enthält hier alle Eigenschaften von `A` und `B`.
Mixed type	`mixed`	Typ für unbekannte Werte, ähnlich zu *any*. Jeder Typ kann `mixed` zugewiesen werden, aber `mixed` kann anderen Typen im Gegensatz zu `any` erst nach Überprüfung der Kompatibilität zugewiesen werden.
Null literal type	`null`	Genau der Wert `null`.
Nullable type (Maybe type)	`?number`	Typ für optionale, möglicherweise undefinierte Werte. Entspricht der Vereinigung aus dem angegeben Typ, `null` und `undefined`.
Number literal type	`42`	Genau dieser numerische Wert.
Number type	`number`	Gleitkommazahlen.

Object type	`{`
	`  [key: string]: number;`
	`  prop: string;`
	`}`
		Typ für Objekte. Durch Notation mit geschweiften Klammern kann der Name und Typ von Attributen spezifiziert werden. Zusätzlich vorhandene Attribute stellen *keine* Typverletzung dar, weil Flow *width subtyping* unterstützt [64]. Mit eckigen Klammern kann eine Index-Signatur angegeben werden. Der Zugriff auf Werte durch beliebige Namen dieses Typs ist damit erlaubt.

Opaque type	`opaque type O = number`	Opake Datentypen sind Typaliase, die ihre zugrunde liegende Implementierung vor dem Benutzer verbergen (*information hiding*).
String literal type	`'literal'`	Genau diese Zeichenkette.
String type	`string`	Zeichenketten.

Basistypen von Flow [58] mit Beispiel.

Basistyp	**Beispiel**	**Kurzbeschreibung**
This type	`this`	Typ für Wert des Schlüsselworts `this` (Selbstreferenz) in Funktionen oder globalem Kontext.
Tuple type	`[Date, number]`	Tupel, also Listen fester Länge mit vorgegebenem Datentyp für jedes Element.
Type alias	`type Type = <FlowType>`	Ermöglicht es, beliebig komplexe Typkonstrukte unter einem neuen Namen, dem Alias, zusammenzufassen.
Type cast	`(variable: string)`	Statische Assertion eines Typs.
Type export	`export type T = number | null`	Export von Typen aus einem Modul.
Type import	`import type T from './types'`	Import von Typen aus anderen Modulen.
Typeof type	`typeof undefined`	Operator, um Flow-Typ eines Werts zu erhalten.
Union type	`number | string | null`	Vereinigungstyp. Hier: Entweder `number` oder `string` oder `null`.
Void type	`void`	Typ für `undefined`. Verwendung zum Beispiel in Funktionen ohne expliziten Rückgabewert.

Hilfstypen

Die Hilfstypen von Flow werden analog zu den Basistypen in Tabelle 2.2 beschrieben. Es gilt anzumerken, dass die drei kursiv markierten Typen *Existential type*, *Subtype* und *Supertype* in der Dokumentation von Flow als überholt (*deprecated*) markiert wurden [62]. Diese sollten daher nicht mehr eingesetzt werden.

Tabelle 2.2: Hilfstypen von Flow [62] mit Beispiel.

Hilfstyp	**Beispiel**	**Kurzbeschreibung**
Call	`$Call<F, T...>`	Berechnet statisch den Typ, der entsteht, wenn der Funktionstyp F mit dem Argument `T` aufgerufen wird. `T` steht dabei für null oder beliebig viele Argumente.
Class	`Class<C>`	Berechnet den Typ (die Klasse) einer Klasseninstanz `C`.
Difference	`$Diff<A, B>`	Berechnet die Differenzmenge zweier Objekttypen `A` und `B`.
Element type	`$ElementType<T, K>`	Berechnet den Typ aller Elemente eines Felds, Tupels oder Objekts, deren Name dem Typ `K` entspricht.

Hilfstypen von Flow [62] mit Beispiel.

Hilfstyp	**Beispiel**	**Kurzbeschreibung**
Exact	`$Exact<O>`	Berechnet die *exakte* Version des Objekttyps `O` (vgl. *Exact object type*).
Existential type	`*`	Spezielle Notation, die Flow anweist, den Typ dieses Ausdrucks (falls möglich) zu inferieren[6] [72].
Keys	`$Keys<O>`	Berechnet den Vereinigungstyp der Attributnamen des Objekttyps `O`.
None maybe type	`$NonMaybeType<T>`	Entfernt die Eigenschaften des *Maybe types*, das heißt es wird ein Typ erzeugt, der alle Werte von `T` außer `null` und `undefined` umfasst.
Object map	`$ObjMap<O, F>`	Berechnet statisch den Typ, der entsteht, wenn der Funktionstyp `F` auf alle Typen der Werte des Objekttyps `O` angewandt wird.
Object map with key	`$ObjMapi<O, F>`	Analog zu *Object map*, jedoch wird in der Abbildung durch `F` neben den Typen der Werte auch die Typen der Namen miteinbezogen.
Property type	`$PropertyType<O, k>`	Berechnet den Typ des Attributnamens `k` eines Objekttyps `O`. `k` muss dabei ein Stringliteral sein.
Read only	`$ReadOnly<O>`	Berechnet den schreibgeschützten Typ des Objekttyps `O`.
Read only array	`$ReadOnlyArray<A>`	Berechnet den schreibgeschützten Typ des Felds `A`.
Rest	`$Rest<A, B>`	Berechnet einen Typ, der dem Ergebnis der Benutzung von JavaScripts Rest-Syntax [42, S. 190] zur Laufzeit entspricht.
Shape	`$Shape<O>`	Berechnet einen Typ, der erlaubt, dass nur eine Untermenge der Attribute des Objekttyps `O` angegeben wird. Deren Typ muss jedoch mit den ursprünglichen Typen der Attribute kompatibel sein.
Tuple map	`$TupleMap<T, F>`	Analog zu *Object map*, jedoch wird der Funktionstyp `F` auf alle Typen der Werte eines Tupels oder Felds angewandt.
Values	`$Values<O>`	Berechnet den Vereinigungstyp der Attributwerte eines Objekttyps `O`.
Subtype	`$Subtype<T>`	Berechnet einen Typ, der nur Subtypen von `T` zulässt (Kovarianz).
Supertype	`$Supertype<T>`	Berechnet einen Typ, der nur Supertypen von `T` zulässt (Kontravarianz).

[6] Dies ist vergleichbar mit dem Schlüsselwort `auto` in C++ [82, S. 151] oder `var` in C# [30].

Typdeklarationen

Tabelle 2.3 zeigt schließlich exemplarisch die Syntax von Typdeklarationen. Für jede externe Bibliothek, die typisiert werden soll, wird hierbei eine eigene Datei angelegt, die das Modul annotiert. Standardmäßig liest Flow derartige Definitionsdateien aus dem Verzeichnis `flow-typed/` im Wurzelverzeichnis eines Projekts ein und bezieht diese bei der Typüberprüfung der gesamten Codebasis mit ein [47].

Tabelle 2.3: Typdeklarationen von Flow [47] mit Beispiel.

Deklaration	**Beispiel**	**Kurzbeschreibung**
Class	`declare class C {}`	Deklaration einer Klasse.
Export	`declare export default () => string`	Deklaration eines Exports aus einem Modul.
Function	`declare function f(number): any`	Deklaration von Funktionen.
Interface	`declare interface I {}`	Deklaration von Schnittstellen.
Module	`declare module 'M' {}`	Deklaration von Modulen.
Type alias	`declare type T = number`	Deklaration von Typaliassen.
Variable	`declare var v: ?string`	Deklaration des Typs von Variablen.

2.2.2 TypeScript

Nachdem das Typsystem von Flow charakterisiert wurde, sollen im Folgenden auch die Prinzipien und der Aufbau von TypeScript erklärt werden, um die Unterschiede der Ansätze zu verdeutlichen. Durch die Gegenüberstellung wird die Argumentation in der Auswertung der Ergebnisse in Kapitel 5 vereinfacht.

Charakterisierung

TypeScript [125] ist eine frei verfügbare, quelloffene Programmiersprache, deren primäres Ziel die Bereitstellung einer statischen Typisierung für JavaScript ist [17, S. 2]. Die Sprache wird von der *Microsoft Corporation* entwickelt und wurde 2012 unter der *Apache License 2.0* [9] veröffentlicht [28]. Die Architektur von TypeScript wurde maßgeblich durch den dänischen Programmierer Anders Hejlsberg entworfen, der bereits unter anderem C# konzipiert hat [74]. Eine formale Spezifikation [125] beschreibt die Grammatik und das Verhalten der Sprache.

In der Publikation „*Understanding TypeScript*“ [17] werden darüber hinaus die theoretischen Konzepte und das Typsystem von TypeScript dargelegt.

Obwohl TypeScript eine unabhängige, vollständige Programmiersprache ist, wurde diese klar mit der Intension entwickelt, möglichst zugänglich für JavaScript-Programmierer zu sein. So stellt die Syntax beispielsweise eine strikte Obermenge von ECMAScript dar [24, S. 25]. Folglich ist jedes JavaScript-Programm auch ein syntaktisch korrektes TypeScript-Programm. Der Funktionsumfang von TypeScript orientiert sich an der fortlaufenden Weiterentwicklung von ECMAScript [17, S. 1]. Erweiterungen von JavaScript werden in der Regel auch in TypeScript integriert. Somit können JavaScript-Entwickler mit Hilfe von TypeScript eine bereits größtenteils bekannte Syntax einsetzen, um Anwendungen statisch zu typisieren.

Durch den TypeScript-Compiler (TSC) wird die Übersetzung von TypeScript-Quelltexten in standardkonformes JavaScript ermöglicht[7]. Dabei werden alle Spuren der statischen Typisierung entfernt, nachdem die Typkorrektheit verifiziert wurde [17, S. 3]. Das Laufzeitverhalten des Programms wird also nicht durch TypeScript beeinflusst, weil die Typen ausschließlich statisch überprüft werden. Der TypeScript-Compiler besitzt eine Vielzahl von Optionen, die unter anderem den Umfang der statischen Analyse, die Einbindung von Standardbibliotheken und die Generierung der JavaScript-Ausgabe beeinflussen [26]. Insbesondere kann hier auch die Striktheit der Typüberprüfungen verschärft werden. Diese sind normalerweise weniger streng als es möglich wäre, um die Einführung von TypeScript in bestehende Projekten zu erleichtern. Standardmäßig können beispielsweise die Werte `null` und `undefined` jedem Typ fehlerfrei zugewiesen werden [26]. Da dies oftmals nicht das erwünschte Verhalten des Typsystems ist, kann eine solche Operation durch Setzen der Option „`strictNullChecks`“ als Typfehler deklariert werden.

Wie ausgeführt legen die Autoren von Flow großen Wert auf die Korrektheit des Typsystems. TypeScript verfolgt hier bewusst einen anderen Ansatz: Es ist ausdrücklich nicht Ziel, ein beweisbares, absolut korrektes System zu entwickeln, sondern es soll eine „Balance zwischen Korrektheit und Produktivität“ [29] gefunden werden. Bierman et al. führen aus, dass auch ein Typsystem, das nicht vollständig korrekt ist, in der Praxis nützlich sein kann, um Programmfehler aufzudecken [17, S. 3]. Ein weiterer Unterschied zu Flow besteht darin, dass TypeScript sämtliche Typen strukturell behandelt [125, S. 38]. Die Autoren begründen diese Entscheidung damit, dass nur eine strukturelle Typisierung der dynamischen Natur von JavaScript gerecht

[7] Seit 2018 kann auch Babel [93] benutzt werden, um TypeScript-Programme in JavaScript zu überführen [109].

würde. So werden zum Beispiel häufig Lambda-Funktionen in JavaScript eingesetzt und Objekte im Allgemeinen nicht durch Klassen, sondern bei Bedarf durch Literale erzeugt [17, S. 3]. Deren Benutzung hängt deshalb nur von ihrem konkreten Aufbau ab, sodass eine strukturelle Betrachtung Sinn ergibt.

Ferner unterstützt TypeScript eine graduelle Typisierung (*gradual typing*), wie sie durch Siek und Taha [114] beschrieben wurde. Hierdurch ist es möglich, nur bestimmte Teilstücke eines Programms zu typisieren, sodass die übrigen Ausdrücke, deren Typ nicht automatisch ermittelt werden kann, implizit den dynamischen Typ *any* erhalten [29]. Wie Flow setzt auch TypeScript Typinferenz ein, um Typen automatisch abzuleiten, wodurch sich die Zahl benötigter Typannotationen reduziert [17, S. 4]. Chaudhuri et al. weisen darauf hin, dass die Typinferenz von TypeScript aber im Gegensatz zu Flow nicht global, sondern nur lokal und an einigen Stellen kontextuell ist [24, S. 24]. Deshalb seien im Allgemeinen bei TypeScript mehr Typannotationen notwendig als bei Flow.

Architektur

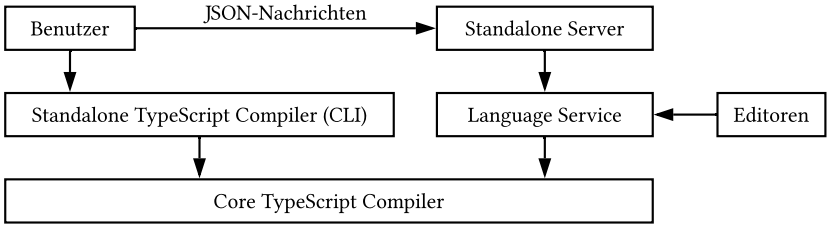


Abbildung 2.1: Überblick über die Architektur von TypeScript nach [94].

Die Architektur TypeScripts kann in mehrere Teile gegliedert werden: Auf unterster Ebene steht der TypeScript-Compiler, der die üblichen Funktionen wie die lexikalische, syntaktische und semantische Analyse des Quelltexts sowie die Codegenerierung umsetzt[8] [94]. Auch die Typüberprüfung wird durch den Compiler innerhalb der semantischen Analyse durchgeführt. Zwei Softwarekomponenten liegen über dem Compiler und greifen auf diesen zu: Einerseits kann die Übersetzung von Quelltexten durch ein Kommandozeilenprogramm (*Standalone TypeScript Compiler*), das die zentrale Benutzerschnittstelle darstellt, angestoßen werden, andererseits wird der „Sprachservice“ (*Language Service*) als Schnittstelle für Editoren bereit gestellt.

[8] Siehe Abschnitt 2.3.1.

Durch den Sprachservice werden Editoren und integrierten Entwicklungsumgebungen Operationen wie Autovervollständigung von Anweisungen, Anzeige von Signaturen oder inferierten Typen, grundlegende Refactoring-Funktionen et cetera ermöglicht [94]. Auf oberster Ebene steht schließlich der TypeScript-Server (*Standalone Server*). Dieser legt die Funktionen des Sprachservices durch ein JSON-basiertes Kommunikationsprotokoll offen, sodass der Service durch andere Anwendungen oder den Benutzer angesprochen werden kann [95].

Beispiel

Entsprechend der Veranschaulichung der Benutzung von Flow soll die Typisierung des Algorithmus der linearen Suche auch durch TypeScript demonstriert werden, um die Typsysteme voneinander abzugrenzen (Quelltext 2.3). Die Quelltexte ähneln sich stark, da sich der Funktionsumfang und die Syntax der Typannotationen von Flow und TypeScript nur in einigen Punkten unterscheidet[9].

```
1  function linearSearch<T>(list: Array<T>, searchValue: T): number | never {
2    for (const [index, value] of list.entries()) {
3      if (value === searchValue) {
4        return index;
5      }
6    }
7    throw new Error('Not found');
8  }
9
10 linearSearch([3, 1, 10, 56], 10);        // 2
11 linearSearch([3, 1, 10, 56], 12);        // Exception (Not found)
12 linearSearch(['foo', 'bar', 'baz'], 3);  // Typfehler (3 ist kein String)
```

Quelltext 2.3: Benutzung von TypeScript anhand des Algorithmus der linearen Suche.

Ein syntaktischer Unterschied liegt beispielsweise beim Schlüsselwort für den leeren Typ. Während dieser in Flow `empty` genannt wird, heißt dieser in TypeScript `never` (Zeile 1). Eine weitere interessante Differenzierung ist das Inferenzverhalten des Typsystems bezüglich des Typparameters `T`. Im Gegensatz zu Flow ist TypeScript im gegebenen Fall in der Lage, den Typ von `T` bei Aufruf der Funktion `linearSearch` in den Zeilen 10 ff. selbstständig abzuleiten, sodass dessen konkreter Typ nicht explizit in spitzen Klammern angegeben werden muss. Auf Grundlage des Typs des ersten Funktionsarguments wird der Typ des zweiten Arguments

[9] Siehe Abschnitt 4.3.2.

durch TypeScript implizit eingeschränkt. Flow inferiert Typparameter generell nicht, sodass diese dort angegeben werden müssen [49]. Die Syntax und Benutzung der Typannotationen im Beispiel ist ansonsten analog zu Flow.

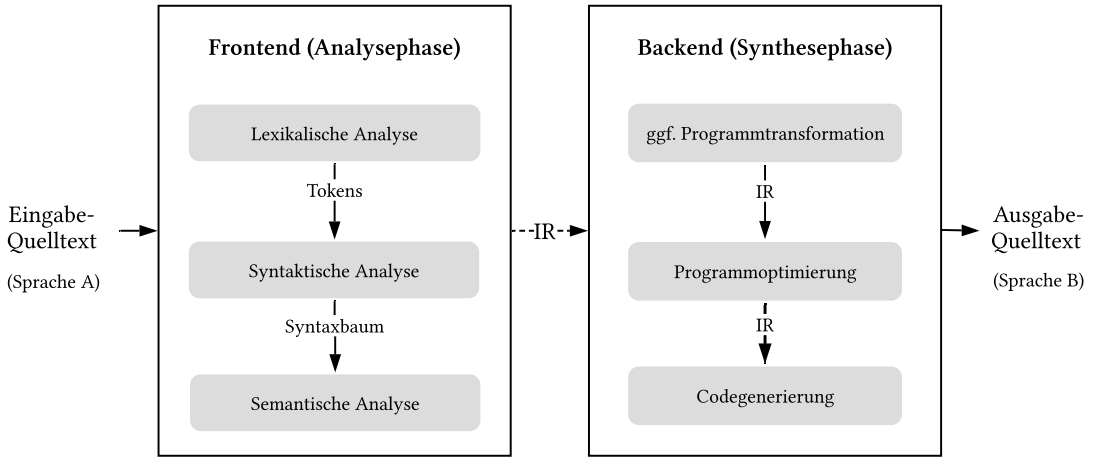
2.3 Transpilierung von Quelltexten

Um die Problemstellung dieser Arbeit, die Übersetzung von Flow nach TypeScript, praktisch zu lösen, soll ein Transpiler umgesetzt werden, welcher die Flow-Typisierung eines Eingabequelltexts in entsprechenden TypeScript-Code transformiert. Bevor dessen Implementierung in Kapitel 4 ausführlich dargelegt wird, werden zunächst die grundlegenden Konzepte und der Aufbau von Transcompilern betrachtet.

2.3.1 Konzepte und Aufbau von Transpilern

Ein Transpiler oder Transcompiler ist ein spezieller Compiler, der den Quelltext einer höheren Programmiersprache in eine andere höhere Programmiersprache übersetzt [6, S. 3]. Anders als bei konventionellen Compilern wird also kein unmittelbar ausführbarer architekturspezifischer Maschinencode erzeugt, sondern der ursprüngliche Quelltext in eine andere Sprache überführt. Auch möglich als Ziel der Transpilierung ist die gleiche Programmiersprache, wenn beispielsweise das Eingabeprogramm entsprechend eines neueren oder älteren Sprachstandards umgeformt werden soll [78]. Abbildung 2.2 zeigt den typischen Aufbau eines Transcompilers. Die Architektur lässt sich wie bei Compilern in zwei Phasen mit mehreren Unterpunkten gliedern: Während die Eingabe im *Frontend* syntaktisch und semantisch analysiert wird, wird das Programm im *Backend* optimiert und der Ausgabequelltext generiert [10, S. 136].

Zunächst wird der Eingabequelltext innerhalb der Analysephase durch den Lexer oder Tokenizer Zeichen für Zeichen eingelesen, um diesen in lexikalisch bedeutungsvolle Zeichenketten, sogenannte *Lexeme*, zu zerlegen [6, S. 43]. Daraufhin werden *Tokens* gebildet, indem jedes dieser Wörter einer syntaktischen Klasse zugeordnet wird. Diese geben die Bedeutung eines Tokens an (zum Beispiel `3` ↦ `INT(3)` oder `!=` ↦ `NEQ`) [123, S. 26]. Die Tokens entsprechen dabei den Terminalsymbolen der formalen Grammatik der Programmiersprache [6, S. 43].



IR: engl. „*intermediate representation*“, also die Transpiler-interne Repräsentation des Programms.

Abbildung 2.2: Architektur eines typischen Transpilers nach [78] und [123, S. 8].

Im zweiten Schritt, der syntaktischen Analyse oder dem *Parsen*, wird anschließend überprüft, ob die Tokenfolge eine Ableitung der kontextfreien Grammatik der Quellsprache darstellen, indem versucht wird, einen entsprechenden *konkreten* Syntaxbaum aufzubauen [111]. Ein Syntax- oder Ableitungsbaum ist ein Graph, der die syntaktische Struktur eines Programms gemäß der zugehörigen Grammatik hierarchisch modelliert. Abbildung 2.3 auf Seite 22 zeigt exemplarisch den *abstrakten* Syntaxbaum (AST) einer simplen JavaScript-Funktion. Ein abstrakter Syntaxbaum ist eine kompaktere Form des konkreten Baums und zeigt lediglich „wesentliche Teile“ [126, S. 21] davon. Falls der Ableitungsbaum nicht erstellt werden kann, so liegt ein Syntaxfehler in der Eingabe vor.

Danach prüft der Transpiler, ob ein gültiges Programm der Eingabesprache vorliegt, indem die statische Semantik analysiert wird [6, S. 8]. Hierunter fällt beispielsweise, dass Variablen in vielen Programmiersprachen vor deren Benutzung zunächst deklariert werden müssen. Um ein solches kontextuelles Programmverhalten zu untersuchen, können Attributgrammatiken verwendet werden [123, S. 161]. Während dieser Phase wird auch die Typkorrektheit überprüft, sofern ein statisches Typsystem vorliegt [111]. Es ist hervorzuheben, dass nicht alle Transcompiler eine semantische Analyse durchführen, sondern manche unmittelbar zum nächsten Schritt übergehen. Folglich werden hier semantische Programmfehler nicht erkannt und einfach übernommen. Zuletzt wird durch das Frontend ein *Zwischencode*[10] des Programms

[10] Engl. *intermediate representation* (IR).

```
function square(x) {
  return x ** 2;
}
```

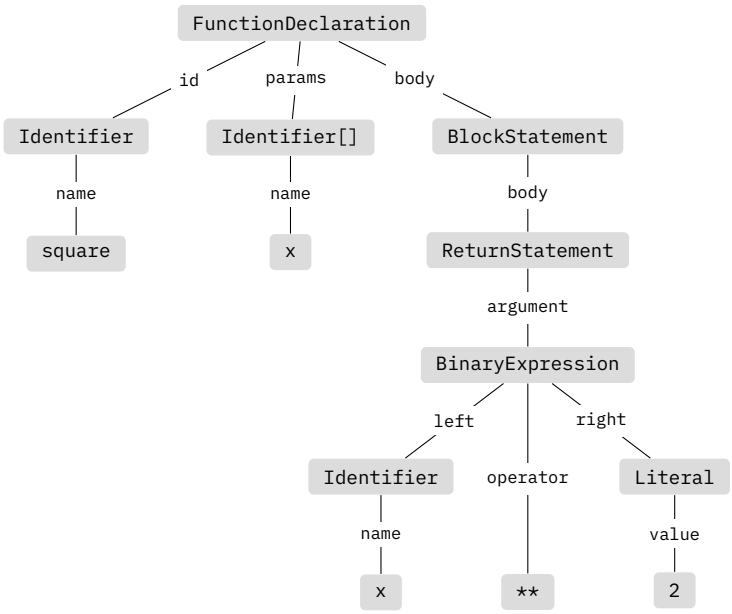


Abbildung 2.3: Abstrakter Syntaxbaum der obigen JavaScript-Funktion gemäß der Spezifikation *ESTree* [75].

erzeugt, der an das Backend übergeben wird. Ein Zwischencode ist im Allgemeinen eine von der Quellsprache und Zielarchitektur unabhängige Datenstruktur innerhalb von Compilern, die das betrachtete Programm modelliert [123, S. 6]. Wie Abschnitt 2.3.2 zeigen wird, verwenden viele Transpiler im JavaScript-Umfeld den abstrakten Syntaxbaum als Zwischencode.

Der erste Schritt der darauffolgenden Synthesephase besteht darin, den Zwischencode gegebenenfalls zu transformieren, um gewünschte Programmeigenschaften herzustellen. Zum Beispiel könnte hier veraltete Syntax durch modernere Notationen ersetzt werden. Als nächstes wird das Programm sprachunabhängig optimiert, das heißt, die Laufzeit oder der Speicherplatzbedarf werden auf Basis einer Programmanalyse reduziert [123, S. 405]. Hierbei können eine Vielzahl unterschiedlicher Transformationen angewandt werden. Möglich ist zum Beispiel die Entfernung nicht erreichbarer Programmstücke (*dead code elimination*) oder die Inline-Ersetzung von Konstanten und Schleifen [123, 111]. Auch hier gilt es zu betonen, dass nicht alle Transcompiler eine derartige Optimierungsphase besitzen.

Schließlich kann im letzten Schritt der endgültige Quelltext in der Zielsprache durch den modifizierten Zwischencode generiert werden [6, S. 505]. Die erzeugte Ausgabe kann dabei dem ursprünglichen Programm ähneln oder völlig andersartig aufgebaut sein, wenn beispielsweise die Namen aller Bezeichner im Quelltext gekürzt wurden, um ein kleineres Programm zu erzeugen (*minification*) [67].

2.3.2 Evaluation bestehender Werkzeuge zur Quelltexttransformation

Im Umfeld von JavaScript sind im Lauf der Jahre eine Vielzahl von Parsern, Codegeneratoren und Transpilern entstanden, welche die Entwicklung weiterer Software wie die des angestrebten Flow-Transcompilers stark vereinfachen können. Weil diese Werkzeuge bereits einen Großteil der benötigten Standardfunktionalität wie das Einlesen und Generieren von Code bereitstellen, muss oftmals nur der Kern des Übersetzers, also im vorliegenden Fall die Transformation der Flow-Typen nach TypeScript, implementiert werden. Im Folgenden sollen einige der relevantesten aktuellen Werkzeuge bezüglich verschiedener Aspekte gegenüber gestellt werden, sodass daraufhin die Entscheidung getroffen werden kann, welcher der Ansätze als Grundlage für die Umsetzung des Transpilers herangezogen wird. Dabei wird nur frei verfügbare, quelloffene Software betrachtet.

Die Auswahl des Werkzeugs hängt entscheidend von der Erfüllung mehrerer technischer Anforderungen ab, die auf Basis der gegebenen Rahmenbedingungen innerhalb des Unternehmens Spreadshirt erarbeitet wurden und in Kapitel 3 ausführlich dargelegt werden. Die erste Vorgabe ist die vollumfängliche Unterstützung aktueller JavaScript-Syntax gemäß der ECMAScript-Spezifikation 2019 [42]. Weiterhin muss die Verarbeitung von *vorläufiger* Syntax durch den Transpiler möglich sein. Darunter werden in dieser Arbeit Erweiterungen von JavaScript verstanden, die noch nicht endgültiger Bestandteil des ECMAScript-Standards sind. Dieser wird kontinuierlich durch das *Technical Committee 39* (TC39) [39] weiterentwickelt. Vorgeschlagene Spracherweiterungen durchlaufen dabei einen mehrstufigen Standardisierungsprozess, der letztlich in die Aufnahme in die Spezifikation münden kann [40]. Derartige Syntax einlesen zu können, ist ebenfalls eine Anforderung, weil diese zum Teil bereits heute in den gegebenen Projekte von Spreadshirt eingesetzt werden[11]. Damit die Übersetzung der Flow-Typen nach TypeScript möglich ist, muss ferner sowohl die Syntax von Flow als auch von

[11] Dies setzt die Benutzung eines Transpilers wie Babel [93] voraus, der solche experimentelle Syntax vor der Auslieferung durch entsprechende Plugins in standardkonformes JavaScript transformiert.

Tabelle 2.4: Vergleich verschiedener Werkzeuge zur Transpilierung von JavaScript-Quelltexten.

Werkzeug	**Typ**	**Format**	**Erw.**	**ES10**	**ES10+**	**Flow**	**TS**	**JSX**	**Aktivität**	**Sterne**	
Acorn	P	ESTree	●	●	◐	○	○	●	5	5.250	[73]
Astring	G	ESTree	●	●	◐	○	○	○	61	500	[18]
Babel	PG	Babel	●	●	●	●	●	●	225	34.650	[93]
Escodegen	G	ESTree	○	○	○	○	○	○	2	1.900	[119]
Esprima	P	ESTree	○	◐	○	○	○	◐	12	5.450	[76]
Recast	PG	diverse	●	●	●	●	●	●	43	2.700	[100]

P	Parser	Erw.	Erweiterbarkeit
G	Codegenerator	ES10	ECMAScript 2019
○	keine Unterstützung	ES10+	vorgeschlagene JavaScript-Erweiterungen
◐	teilweise Unterstützung	TS	TypeScript
●	vollständige Unterstützung	Sterne	Anzahl der Sterne auf GitHub

Stand: Oktober 2019

Aktivität: Entspricht der Summe der Zahl von *Merge Requests*, neuer bzw. geschlossener Fehlerberichte, veröffentlichter Git-Commits auf dem Hauptzweig und der Anzahl beteiligter Autoren innerhalb eines Monats auf der Plattform GitHub [122].

TypeScript vollständig unterstützt werden. Darüber hinaus muss sogenannte JSX-Syntax [52] (*JavaScript XML*) eingelesen werden können, weil die zu migrierenden Projekte auch diese Notation einsetzen. Die letzte Anforderung ist schließlich, dass auf Grundlage des transformierten Programms entsprechender TypeScript-Code generiert werden kann.

Die Eigenschaften der in Betracht kommenden Werkzeuge werden in Tabelle 2.4 aufgelistet. Dabei werden verschiedene Parser (*Acorn*, *Esprima*), Codegeneratoren (*Astring*, *Escodegen*) und ganze Transpiler (*Babel*, *Recast*) für JavaScript betrachtet. Jedes der Werkzeuge benutzt intern den abstrakten Syntaxbaum der Eingabe als Zwischencode. Da außer Babel alle Alternativen die Spezifikation *ESTree* für den Aufbau des abstrakten Syntaxbaums verwenden, können diese Parser und Codegeneratoren prinzipiell beliebig kombiniert werden, um einen Transcompiler zusammenzusetzen. Der ESTree-Standard wird von Mitgliedern verschiedener Projekte im Umfeld der statischen Analyse von JavaScript stetig weiterentwickelt und spezifiziert den Aufbau abstrakter Syntaxbäume von ECMAScript-Programmen [75]. Weil aber weder Astring noch Escodegen TypeScript als Ausgabesprache unterstützen, kann dieser Ansatz nicht verfolgt werden, um die gegebene Problemstellung zu lösen.

Nach Kenntnisstand des Autors existieren außer Babel und Recast keine weiteren Werkzeuge, die einen Codegenerator bereitstellen, der TypeScript auf Grundlage von JavaScript-Syntaxbäumen erzeugen kann. Auch sind diese Projekte die einzigen der betrachteten, die aktuelle und vorläufige JavaScript-Syntax sowie Flow, TypeScript und JSX vollständig verarbeiten können. Jedoch wird dies in Recast nur bei Verwendung des Parsers von Babel [12] erzielt. Recast setzt standardmäßig Esprima als Parser ein, aber es können auch andere Parser wie Babel verwendet werden, um nicht standardkonforme Syntaxerweiterungen von JavaScript einzulesen. Ein interessantes Merkmal von Recast ist, dass hier die Formatierung des ursprünglichen Quelltexts konsistent beibehalten wird, indem die abstrakten Syntaxbäume der Ein- und Ausgabe verglichen werden, sodass daraufhin nur der Code der geänderten Teilbäume neu generiert wird [100]. Dieser Ansatz wirkt zunächst aussichtsreich, weil so vermeintlich die originalgetreue Formatierung der TypeScript-Ausgabe erreicht werden kann. Jedoch hat sich experimentell gezeigt, dass der Programmierstil der modifizierten und damit neu erstellten Programmteile, anders als von Recast behauptet, zum Teil doch vom ursprünglichen Code abweicht. Die Formatierung unveränderter Abschnitte wird aber tatsächlich präzise beibehalten. Auch wurde festgestellt, dass Kommentare innerhalb der modifizierten Ausdrücke komplett verloren gehen, was für die Erzielung einer korrekten Übersetzung inakzeptabel ist.

Wie die Tabelle zeigt, ist Babel zudem im Vergleich zu Recast deutlich verbreiteter und wird aktiver entwickelt. Deshalb kann Babel als die ausgereiftere Software betrachtet werden. Weiterhin bietet Babel eine gute Erweiterbarkeit durch ein Plugin-System [89] und besitzt eine umfangreiche Dokumentation [127]. Aufgrund dieser Vorteile und Alleinstellungsmerkmale wird Babel als Grundlage für die Implementierung gewählt.

2.3.3 Babel

Funktionsweise

Zur Erleichterung des Verständnisses der Ausführung des umgesetzten Flow-Transpilers in Kapitel 4 soll zunächst die grundsätzliche Funktionsweise von Babel umrissen werden. Die Ausführung von Babel gliedert sich in folgende drei Phasen [89]. Diese sind in weiten Teilen analog zu dem beschriebenen Aufbau eines typischen Transcompilers.

1. **Parsen des Eingabecodes**
   Zunächst wird der Eingabequelltext in zwei Schritten eingelesen, um den abstrakten Syntaxbaum des Programms zu erzeugen: Als Erstes wird der Code während der lexikalischen Analyse mittels des Lexers in Tokens zerlegt. Anschließend werden diese in der syntaktischen Analyse zu einer Datenstruktur umgeformt, die den zugehörigen abstrakten Syntaxbaum repräsentiert. Jedem Knoten des Baums wird dabei ein eindeutiger Tokentyp zugewiesen, der dessen syntaktische Bedeutung widerspiegelt.

2. **Transformation des Programms**
   Während der zweiten Phase wird daraufhin die Kernfunktionalität von Babel, die Programmtransformation umgesetzt: Dabei wird der abstrakte Syntaxbaum durch das *Besucher*-Entwurfsmuster[12] rekursiv traversiert und die Knoten des Baums sukzessive modifiziert, gelöscht bzw. neu erstellte Elemente eingefügt. Das Entwurfsmuster beschreibt, wie ein Algorithmus auf einer komplexen Objektdatenstruktur, unabhängig von der konkreten Implementierung der zugrunde liegenden Klassen, ausgeführt werden kann [68, S. 634 f.]. Im vorliegenden Fall ermöglicht die Anwendung die individuelle Adressierung einer bestimmten Untermenge der Knoten des Syntaxbaums, sodass dort die gewünschte Transformation des Programms durchgeführt werden kann.

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

Während der Ausführung der Implementierung des Transpilers in Abschnitt 4.3 wird hauptsächlich die zweite Phase betrachtet, da hier die vorliegende Problemstellung, die Transformation der Flow-Typannotationen nach TypeScript, umgesetzt wird. Das Parsen des Eingabequelltexts und das Generieren der Ausgabe kann durch Verwendung der gegebenen Bibliotheksfunktionen von Babel simpel realisiert werden und bedarf keiner genaueren Untersuchung.

[12] Das Besucher-Entwurfsmustern (engl. *visitor pattern*) gehört zu den 23 Entwurfsmustern, die im Standardwerk „*Design Patterns: Elements of Reusable Object-Oriented Software*“ der „*Gang of Four*“ (E. Gamma, R. Helm, R. Johnson und J. Vlissides) beschrieben werden [69, S. 306 ff.].

Babel-Plugins

Da die entscheidende Phase der Transpilierung, die Programmtransformation, bei Babel durch Plugins erzielt wird, sollen diese genauer betrachtet werden. Plugins sind die elementaren Bausteine, die eine flexible Erweiterung von Babel ermöglichen. Selbst der Kern des Transcompilers ist aus einer Vielzahl von Standard-Plugins zusammengesetzt, die in ihrer Gesamtheit die Funktionalität des Systems abbilden [93]. Dies verdeutlicht die tiefgreifende Modularität der Architektur von Babel.

Ein Babel-Plugin ist eine JavaScript-Funktion, welche gemäß der vorgegebenen Schnittstellen ein Objekt mit verschiedenen Attributen zurückliefern muss. Mindestens anzugeben ist dabei lediglich die Abbildung der gewünschten Knotentypen des abstrakten Syntaxbaums auf Besucherfunktionen [89]. Deren Implementierung setzt die angestrebte Quelltexttransformation um. Es ist in der Praxis gängig, mehrere Plugins einzusetzen, sodass ein Knoten während der Verarbeitung durch Babel eine Reihe unabhängiger Transformationen durchlaufen kann. Hierdurch kann die erwünschte Transpilierung von JavaScript-Quelltexten flexibel durch Kombination vieler kleiner Bausteine realisiert werden. Auch möglich ist die Angabe einer hierarchischen Abhängigkeitsstruktur, sodass die Verwendung eines Plugins zur impliziten Aktivierung weiterer Plugins führt [89].

```
1  // var foo => var FOO
2  module.exports = function() {
3    return {
4      visitor: {
5        Identifier(path) {
6          path.node.name = path.node.name.toUpperCase();
7        }
8      }
9    };
10 };
```

Quelltext 2.4: Minimalbeispiel eines Babel-Plugins: Die Namen aller Bezeichner (`Identifier`) werden in Großbuchstaben umgewandelt.

Der konkrete Aufbau von Babel-Plugins soll durch ein Minimalbeispiel gezeigt werden. Quelltext 2.4 zeigt den Code eines simplen, aber vollständigen Plugins, welches lediglich den Namen aller Bezeichner (`Identifier`) eines JavaScript-Programms in Großbuchstaben setzt. Hierfür wird eine gleichnamige Besucherfunktion für den Knotentyp `Identifier` definiert (Zeile 5). Diese erhält den *Pfad* der so adressierten Identifier-Knoten als Argument und kann diesen

wie gewünscht transformieren. Der Pfad eines AST-Knotens ist ein Objekt, das die Beziehung des Knotens zu seinen Elternelementen modelliert und diesen um Metainformationen anreichert [89]. Es enthält eine Vielzahl von Methoden, mittels derer der Pfad und der Syntaxbaum als Ganzes manipuliert werden kann.

Alle Knotentypen des abstrakten Syntaxbaums von Babel werden einerseits in der Spezifikation des Parsers [92, 12], andererseits in der Dokumentation der Bibliothek `@babel/types` [13] beschrieben. Der Aufbau des von Babel eingesetzten abstrakten Syntaxbaums liegt in einem eigenen Format vor, das auf dem ESTree-Standard [75] basiert. Mit der sechsten Version von Babel wurde entschieden, von der ESTree-Spezifikation abzuweichen, damit auch vorläufige JavaScript-Erweiterungen unterstützen werden können [132].

3 Ziel- und Anforderungsanalyse

Nachdem die Grundlagen der Thematik erörtert wurden, werden nachfolgend die Ziele der Migration von Flow nach TypeScript innerhalb des Unternehmens Spreadshirt und die Anforderung an die Implementierung des Transpilers definiert. Zunächst soll jedoch die Ausgangslage zu Beginn dieser Arbeit beschrieben werden, um die Rahmenbedingungen zu verdeutlichen.

3.1 Ausgangslage und Rahmenbedingungen

3.1.1 Kurzvorstellung der Projekte von TeamShirts

Das Leipziger Unternehmen *sprd.net AG* (*Spreadshirt*) ist ein seit 2002 bestehender Anbieter verschiedener eCommerce-Plattformen, welche die individuelle On-Demand-Bedruckung von Kleidung und Accessoires ermöglichen [5]. Die Produkte können dabei von den Kunden online durch vorgegebene oder eigene Motive gestaltet und anschließend bestellt werden. Um weitere Zielgruppen wie Sportmannschaften, Vereine, Belegschaften usw. besser anzusprechen, wurde 2014 der Geschäftsbereich *TeamShirts* gegründet [4], in dessen Kontext diese Arbeit entstanden ist. TeamShirts betreibt verschiedene Webanwendungen, deren Frontend vorrangig auf JavaScript basiert. Innerhalb des Unternehmens gibt es strategische Überlegungen, die bestehenden Projekte nach TypeScript zu migrieren, um die derzeitige Typisierung mit Flow so zu ersetzen. Die Gründe hierfür werden in Abschnitt 3.2 ausführlich dargelegt. Der Wechsel zu TypeScript wurde durch den in dieser Arbeit entworfenen und realisierten Transpiler für zwei dieser Projekte umgesetzt. Um die Nachvollziehbarkeit der weiteren Ausführungen zu erleichtern, sollen diese kurz vorgestellt werden.

Components Alle Frontend-Projekte von TeamShirts bauen auf der Softwarebibliothek *React* [57] auf, welche die Programmierung von Benutzeroberflächen auf Basis von sogenannten *Komponenten*[13] ermöglicht [116]. Diese stellen wiederverwendbare, voneinander

[13] React-Komponenten sind nicht mit *Webkomponenten* [37] zu verwechseln. Diese verfolgen ein ähnliches, aber nicht identisches Konzept. React interagiert beispielsweise im Gegensatz zu Webkomponenten direkt mit dem *Document Object Model* (DOM) des HTML-Dokuments [63] statt mit dem sogenannten *Shadow DOM*.

unabhängige Elemente dar, die aus HTML, CSS und JavaScript aufgebaut sind und durch Komposition zu komplexeren Strukturen zusammengesetzt werden können. Das erste Projekt von TeamShirts *Components* stellt eine Bibliothek derartiger React-Komponenten dar, die abteilungsintern in verschiedenen anderen Softwaremodulen eingesetzt wird.

Helios *Helios* ist ein Service für *Server-Side Rendering* (SSR). Unter *Server-Side Rendering* versteht man die Generierung des gesamten Quelltexts einer Webseite durch einen Server. Dies ist insbesondere für Webanwendungen interessant, deren Aufbau normalerweise clientseitig mittels JavaScript realisiert wird, da sich auf diese Weise verschiedene Performance-Vorteile ergeben. So wird beispielsweise eine schnellere Interaktivität[14] der Seite erreicht, da durch SSR weniger Berechnungen im Webbrowser nötig sind [96]. TeamShirts verwendet Helios, um Seiten, die durch verschiedene React-Komponenten aufgebaut sind, serverseitig zu erstellen und auszuliefern.

3.1.2 Evaluation bestehender Ansätze zur Transpilierung von Flow

Vor Beginn der Arbeit wurde evaluiert, ob bereits Lösungsansätze für die gegebene Problemstellung, die Transpilierung von Flow-Typen nach TypeScript, bestehen. Tatsächlich existierten zum damaligen Zeitpunkt im Februar 2019 zwei Projekte auf der Software-Plattform GitHub, welche das gleiche Ziel wie diese Arbeit verfolgen: Bereits im November 2017 begann der Programmierer Boris Cherny mit der Entwicklung eines Babel-Plugins zur Transpilierung von Flow [25]. Weiterhin ist im April 2018 ein vergleichbares Projekt durch Yuichiro Kikura entstanden [87]. Nach näherer Betrachtung und Erprobung der zwei Plugins wurde jedoch rasch festgestellt, dass sich beide Werkzeuge noch in einem frühen Stadium befinden und deren Funktionsumfang unvollständig ist. Da die Weiterentwicklung der Plugins darüber hinaus nur sporadisch vorangetrieben wurde, wurde angenommen, dass die Projekte inaktiv sind. Eine korrekte, ganzheitliche Übersetzung der vorliegenden JavaScript-Quelltexte von TeamShirts war somit nicht umsetzbar. Folglich wurde beschlossen, selbst einen Transpiler für Flow zu entwickeln, der allen in Abschnitt 3.3 ausgeführten Anforderungen gerecht wird.

Einige Monate später im Sommer 2019 ist die Arbeit von Yuichiro Kikura an dessen Plugin wieder aufgenommen worden und es ist ein drittes Projekt von Kevin Barabash, Mitarbeiter

[14] Engl. „*time to interactive*“ (TTI).

der eLearning-Plattform *Khan Academy* [3], hinzu gekommen [14]. Das etwa zeitgleiche Aufkommen mehrerer Projekte mit der gleichen Zielsetzung wie diese Arbeit kann als Beleg dafür betrachtet werden, dass ein großer Bedarf zu existieren scheint die vorliegende Problemstellung zu lösen. Der in dieser Arbeit umgesetzte Transpiler wird in Kapitel 5 den konkurrierenden Ansätzen bezüglich verschiedener Aspekte gegenüber gestellt, sodass die erreichten Ergebnisse durch den Vergleich besser eingeordnet werden können.

3.2 Ziele der Migration zu TypeScript

In Zusammenarbeit mit den Softwareentwicklern bei TeamShirts wurden die folgenden vier Ziele der Migration von Flow nach TypeScript definiert. Diese und die im darauffolgenden Abschnitt dargelegten Anforderungen an den Transpiler bilden die Grundlage für die kritische Bewertung der Ergebnisse in Kapitel 5.

3.2.1 Erkennung weiterer Typ- und Programmfehler

Die Hauptmotivation für den Wechsel des Typsystems bei TeamShirts ist die Erkennung weiterer Typ- und Programmfehler im bestehenden und zukünftigen Code der Frontend-Projekte. Es wird angenommen, dass TypeScript hier, insbesondere auch durch die im nächsten Abschnitt beschriebene Unterstützung externer Bibliotheken, Flow überlegen ist und damit langfristig mehr Typfehler aufgedeckt werden können.

Im Umfeld einer eCommerce-Plattform sind JavaScript-Laufzeitfehler in der Produktivumgebung fatal, da diese zum Beispiel im vorliegenden Fall bei TeamShirts Bestellabschlüsse verhindern können und dies zu Umsatzeinbußen führt. Folglich besteht ein starkes Interesse des Unternehmens, die Qualität und Robustheit der betriebenen Software weiter zu steigern, indem bestehende Mängel behoben und zukünftige Programmfehler vermieden werden. Es gilt daher in der Auswertung zu zeigen, dass TypeScript in der Lage ist, tatsächliche Programmfehler im bestehenden Quelltext durch Typverletzungen aufzudecken.

3.2.2 Unterstützung externer Bibliotheken

Da alle Projekte von TeamShirts auf einer Vielzahl externer Softwarebibliotheken wie React [57] et cetera aufbauen, ist die umfassende Unterstützung dieser durch das Typsystem eine wichtige Zielvorgabe. Sowohl Flow als auch TypeScript ermöglichen es, den Funktionsumfang und die Struktur von Bibliotheken durch Typdeklarationen in Definitionsdateien zu beschreiben[15]. Hierdurch können Module, die selbst keine Typdeklarationen mitliefern, nachträglich um eine Typisierung in Flow oder TypeScript erweitert werden, sodass die Vorteile statischer Typsysteme auch hier genutzt werden können. Das Risiko, dass Bibliotheken inkorrekt verwendet werden, kann so erheblich reduziert werden, weil zum Beispiel fehlerhafte Funktionsaufrufe durch das Typsystem unmittelbar erkannt werden.

Da es aufwändig und fehleranfällig ist selbst korrekte Typdeklarationen für fremde Bibliotheken anzufertigen, ist es für deren tiefgreifende Unterstützung durch das Typsystem ausschlaggebend, dass bereits qualitativ hochwertige Typisierungen extern vorliegen. Dies bedeutet, dass die Definitionen möglichst vollständig und aktuell hinsichtlich der Softwareversion des Pakets sind. Einige Bibliotheken stellen selbst Deklarationsdateien bereit, für andere existieren separate Typisierungen, die durch die Entwicklergemeinschaft fortlaufend aktualisiert werden. Es besteht die Vermutung, dass bei TypeScript im Vergleich zu Flow Typdeklarationen für eine insgesamt größere Zahl von Bibliotheken bestehen und diese in mindestens gleichwertiger Qualität vorliegen. Falls sich diese These als tatsächlich haltbar erweist, ist das Ziel erreicht.

3.2.3 Performance der Typüberprüfungen

Ein weiterer Aspekt, der durch die Migration zu TypeScript verbessert werden soll, ist die Performance, das heißt, die Schnelligkeit der Typüberprüfungen. Üblicherweise wird der Flow- bzw. TypeScript-Sprachserver, der das gesamte Projekt und insbesondere offene Dateien im Hintergrund kontinuierlich auf Typverletzungen überprüft, durch den Editor oder die integrierte Entwicklungsumgebung gestartet. Es ist entscheidend, dass die Ergebnisse dieser Berechnung der Typkorrektheit möglichst schnell vorliegen, damit etwaige Fehler unmittelbar im Editor angezeigt werden können. Falls diese stark verzögert erst nach einigen Sekunden ausgegeben werden, so verlangsamt dies den Workflow des Programmierers und beeinträchtigt dessen

[15] Vgl. Abschnitt 2.2.1.

effizientes Arbeiten. Neben diesen inkrementellen Überprüfungen ist auch die Laufzeit der vollständige Berechnung der Typkorrektheit bedeutsam, da derartige Tests beispielsweise im Zuge der kontinuierlichen Softwareintegration ausgeführt werden. Ein weiteres Erfolgskriterium des Wechsels zu TypeScript ist damit die Erzielung einer mindestens gleichwertigen Performance der Typüberprüfungen.

3.2.4 Zukunftssicherheit und Transparenz der Technologie

Zuletzt ist einerseits die Zukunftssicherheit, andererseits die Transparenz der eingesetzten Technologie von Bedeutung. Hierunter werden innerhalb dieser Arbeit verschiedene Fragestellungen verstanden: Existiert ein öffentlich einsehbarer Projektplan (*Roadmap*) für die Weiterentwicklung des Systems? Wie gesichert ist die langfristige Unterstützung des Projekts durch die ursprünglichen Autoren[16] der Technologie? Werden strategische Entscheidungen öffentlich kommuniziert? Wird die Software quelloffen oder proprietär entwickelt? Werden gemeldete Programmfehler innerhalb einer angemessenen Zeitspanne behoben?

All diesen Fragen gemein ist das Bedürfnis der Benutzer nach Stabilität, Transparenz und Verlässlichkeit der eingesetzten Technologie. Das Entwicklerteam von TeamShirts hat den Eindruck gewonnen, dass die Entwicklung von Flow oftmals intransparent voran getrieben wird, sodass eine gewisse Verunsicherung bezüglich der Zukunft der Technologie besteht. Die Autoren von Flow räumen im 2019 veröffentlichten Artikel „*What the Flow team has been up to*“ [23] selbst ein, dass die Kommunikation der Pläne für Flow zuletzt ungenügend war. Weil die Softwareprojekte bei TeamShirts für gewöhnlich über mehrere Jahre hinweg erweitert und gewartet werden, ist die Beständigkeit und die kontinuierliche Weiterentwicklung des eingesetzten Typsystems von großer Bedeutung für das Unternehmen. Das letzte Ziel der Migration ist deshalb die Steigerung der Zukunftssicherheit und Transparenz durch den Wechsel zu TypeScript.

[16] *Facebook Inc.* (Flow) [24] bzw. *Microsoft Corporation* (TypeScript) [125].

3.3 Technische Anforderungen an den Transpiler

Neben den ausgeführten Zielen wurden weiterhin fünf technische Anforderungen an den Transpiler erarbeitet, welche die Grundlage für die spätere Bewertung der Implementierung bilden. Um sicherzustellen, dass das Werkzeug möglichst generisch einsetzbar ist, wurden die Anforderungen nur wo nötig speziell auf die Gegebenheiten bei TeamShirts zugeschnitten.

3.3.1 Äquivalente und vollständige Übersetzung der Flow-Typen

Die bedeutsamste Anforderung an den Transpiler ist die äquivalente Übersetzung sämtlicher Flow-Sprachelemente nach TypeScript. Durch die Forderung der Vollständigkeit wird erreicht, dass jedes beliebige Flow-Eingabeprogramm transpiliert werden kann. Da durch die Spezifikation des abstrakten Syntaxbaums von Babel [92] klar definiert ist, welche der AST-Knoten Flow-Syntax darstellen, ist der Umfang einer vollständigen Implementierung exakt eingegrenzt. Jedoch werden einige der Funktionen von Flow wie beispielsweise opake Typen in TypeScript nicht unterstützt [118], sodass eine absolut bedeutungsgleiche Übersetzung dieser Typen unmöglich ist. Der Verlust von Typinformation wird deshalb für diese Transformationen akzeptiert, allerdings soll der Benutzer bei Auftreten dieser Fälle durch eine aussagekräftige Warnung benachrichtigt werden.

3.3.2 Semantisch äquivalente Transpilierung des Quelltexts

Darüber hinaus muss gewährleistet werden, dass die Semantik des ursprünglichen JavaScript-Programms durch den Transpilierungsprozess nicht verändert wird, um sicherzustellen, dass keine subtilen semantischen Fehler in den resultierenden TypeScript-Code eingeschleust werden. Die *Wirkung* der Gesamtheit aller Unterprogramme muss also vor und nach der Migration identisch sein. Ausgenommen hiervon sind die im vorherigen Abschnitt angeführten Flow-Typen, die von TypeScript nicht unterstützt werden und daher nicht vollkommen äquivalent übersetzt werden können.

3.3.3 Unterstützung aktueller und vorläufiger JavaScript- sowie JSX-Syntax

Aufgrund der kontinuierlichen Weiterentwicklung JavaScripts ist die Syntax der Sprache in den letzten Jahren wiederholt um neue Elemente erweitert worden. Beispielsweise wurden 2017 die Schlüsselworte `async` und `await` eingeführt, welche die asynchrone Programmierung in JavaScript erleichtern [41, S. 430]. Damit eine universelle Übersetzung beliebiger Eingaben umsetzbar ist, ist es entscheidend, dass der Transpiler jegliche standardkonforme JavaScript-Syntax gemäß der aktuellen ECMAScript-Spezifikation 2019 [42] verarbeiten kann. Auch die Unterstützung vorläufiger, noch nicht endgültig standardisierter Spracherweiterungen stellt eine wesentliche Anforderung dar, weil diese im Umfeld einiger populärer Bibliotheken bereits heute verwendet werden. So wird zum Beispiel die Erweiterung „*Class field declarations for JavaScript*“ [45] häufig von React-Programmierern benutzt, da deren Syntax die Verwendung mancher Funktionen der Bibliothek vereinfacht [50]. Weil alle Frontend-Projekte von TeamShirts, wie ausgeführt, auf React basieren, ist darüber hinaus die Unterstützung von JSX-Syntax [52] eine wesentliche Anforderung an den Transpiler. JSX (*JavaScript XML*) ist eine von React eingeführte syntaktische Erweiterung von JavaScript, die dazu verwendet wird den HTML-Aufbau von Komponenten anzugeben.

3.3.4 Verarbeitung gesamter Projektverzeichnisse

Eine umfangreiche Codebasis besteht im Normalfall aus Hunderten von Einzeldateien. Tabelle 3.1 auf Seite 36 gibt einen Überblick über die konkrete Anzahl und Zusammensetzung der JavaScript-Dateien im vorliegenden Fall bei TeamShirts. Um alle Dateien eines Projekts sukzessive übersetzen zu können, soll eine Stapelverarbeitung implementiert werden, welche das rekursive Einlesen und Verarbeiten eines gesamten Verzeichnisses realisiert. Dabei soll es auch möglich sein, nur bestimmte Dateien in Unterverzeichnissen ein- bzw. auszuschließen, sodass die Menge der Eingabedateien flexibel eingegrenzt werden kann.

3.3.5 Beibehaltung der Quelltextformatierung

Als letzte Anforderung wurde schließlich definiert, dass die Quelltextformatierung der Projekte nach Ausführung des Transpilers so originalgetreu wie möglich beibehalten werden muss,

Tabelle 3.1: Anzahl von JavaScript-Dateien und Verteilung zugehöriger Leer-, Kommentar- und Codezeilen der zwei Projekte von TeamShirts.

Projekt	**Dateien**	**Leerzeilen**	**Kommentarzeilen**	**Codezeilen**
Components	331	4.341	963	24.936
Helios	353	4.814	495	40.127

weil der Programmierstil bedeutsam für die Wartbarkeit und Verständlichkeit von Software ist [86, S. 146]. Darüber hinaus bestehen teaminterne Absprachen bezüglich des Aufbaus und der Formatierung des Codes, welche durch die Migration nicht verworfen werden dürfen. Unter der Formatierung wird nachfolgend die Einrückung des Codes, Zeilenumbrüche und die Position der Leerzeichen und -zeilen verstanden. Auch Block- und Zeilenkommentare müssen korrekt in die TypeScript-Ausgabe übernommen werden.

Besonders hervorzuheben ist hierbei die Positionierung von speziellen Kommentaren, die verwendet werden, um Code-Fragmente von den Überprüfungen durch *ESLint* [130] auszunehmen. ESLint ist ein Werkzeug zur statischen Analyse von JavaScript- und TypeScript-Quelltexten, das die Einhaltung eines festgelegten Programmierstil ermöglicht, indem Ausdrücke, die eine dieser Regeln verletzen, offengelegt werden.

4 Umsetzung

Nachdem die Ziele der angestrebten TypeScript-Migration charakterisiert und die Anforderungen an den geplanten Transpiler zur Übersetzung von Flow nach TypeScript ausgeführt wurden, soll im Folgenden der Entwurf und die Details der Implementierung umfassend betrachtet werden. Auf Basis der Gegenüberstellung verschiedener Werkzeuge zur Transformation von JavaScript-Quelltexten in Abschnitt 2.3.2 wurde Babel [93] als Grundlage der vorliegenden Umsetzung gewählt.

4.1 Softwarearchitektur

Mit der Entscheidung, den Übersetzer als Babel-Plugin zu implementieren, ist dessen Grundarchitektur bereits in Teilen festgelegt, da alle Plugins die vorgegebenen Programmschnittstellen von Babel erfüllen müssen. Bevor auf Einzelheiten der Umsetzung näher eingegangen wird, soll zunächst der grundsätzliche konzeptionelle Aufbau der Anwendung skizziert werden. Abbildung 4.1 verschafft einen Überblick über die verschiedenen Komponenten des Systems und deren Beziehung zueinander.

Die Architektur gliedert sich in zwei Teile: Ein Kommandozeilenprogramm stellt die Benutzerschnittstelle dar, welche die Eingabeverzeichnisse bzw. -dateien als Argument entgegen nimmt und verschiedene Optionen bereitstellt, um das Verhalten der Übersetzung zu beeinflussen[17]. Die zweite Komponente ist das Babel-Plugin, das die Transpilierung des Flow-Codes nach TypeScript realisiert. Das Kommandozeilenprogramm liest sukzessive alle Eingabeverzeichnisse bzw. -dateien ein und startet intern die Übersetzung des Quelltexts durch Babel. Hierfür wird das umgesetzte Plugin geladen und dieses auf die Eingabe angewendet. Danach kann der TypeScript-Code generiert, formatiert und in Dateien oder auf die Standardausgabe geschrieben werden.

[17] Siehe Tabelle 4.5.

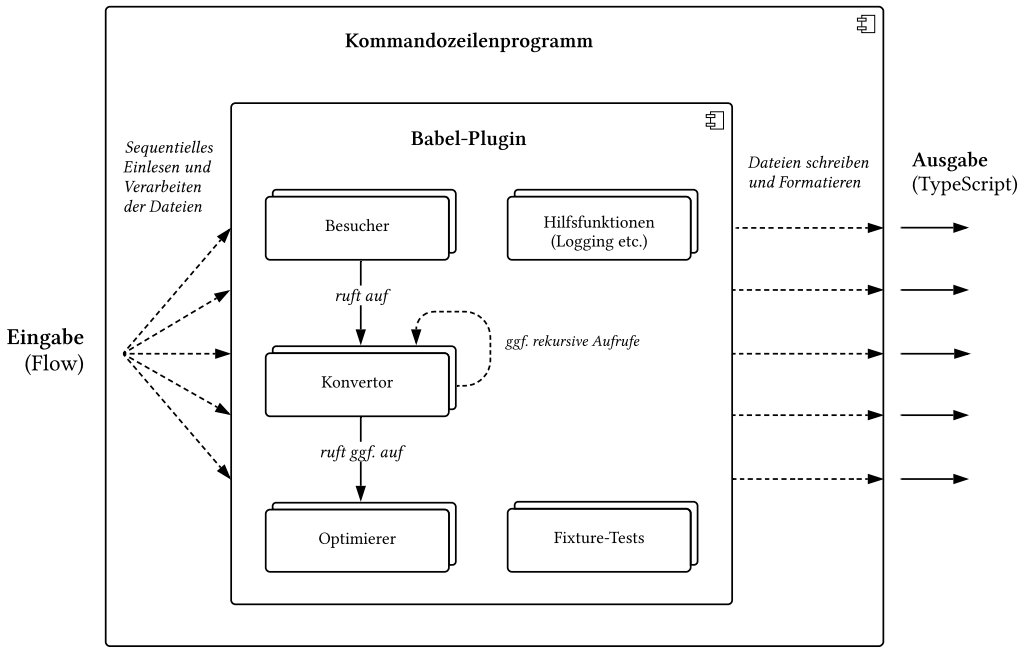


Abbildung 4.1: Überblick über die Komponenten des Transpilers.

Das Plugin setzt sich aus verschiedenen Modulen zusammen: Auf oberster Ebene befinden sich die Besucherfunktionen[18], welche die eigentliche Programmtransformation realisieren. Diese adressieren alle Pfade des abstrakten Syntaxbaums der Eingabe, die Flow-Syntax darstellen. Bei der Traversierung des Baums durch Babel werden so alle zugehörigen Knoten der Pfade durch Ausführung verschiedener *Konverter* in äquivalente TypeScript-Ausdrücke übersetzt. Dabei kann es während der Verarbeitung zu rekursiven Aufrufen weiterer Besucher bzw. Konverter kommen. In einigen Fällen liegen weiterhin Methoden zur Optimierung des Transformationsresultats vor, die nach der Konvertierung gegebenenfalls angewandt werden. Darüber hinaus beinhaltet das Plugin Hilfsfunktionen, um verschiedene Aufgaben wie die Ausgabe von Fehlern und Warnungen umzusetzen. Zuletzt enthält das Plugin eine Vielzahl von Modultests (*Unit tests*), welche die korrekte Funktionalität aller Komponenten überprüfen.

[18] Vgl. Abschnitt 2.3.3.

4.2 Entwicklungsprozess

Bevor die Implementierung der Programmtransformation im Detail betrachtet wird, sollen zunächst zwei Aspekte des Entwicklungsprozesses dargelegt werden, die diesen wesentlich unterstützt haben.

4.2.1 Testgetriebene Entwicklung

Die korrekte Übersetzung der Flow-Typen nach TypeScript ist, wie ausgeführt, die wichtigste Anforderung an den Transpiler[19]. Essentiell ist daher die Bereitstellung zuverlässiger Testmechanismen, um Softwareregressionen während der Entwicklungphase frühzeitig festzustellen. Unter Regressionen werden Programmfehler verstanden, die nach bestimmten Ereignissen wie der Implementierung weiterer Funktionen oder Software-Upgrades unvorhergesehen in bereits getesteten Modulen auftreten [101, S. 218]. Zur Erfüllung der Anforderung wurde der Ansatz der testgetriebenen Entwicklung[20] gewählt, um die korrekte Funktionalität und Wechselwirkung aller Bestandteile des Transpilers kontinuierlich zu überprüfen. Die testgetriebene Entwicklung hat ihren Ursprung im Vorgehensmodell *Extreme Programming* [83] aus der Softwareentwicklung und sieht im Gegensatz zu klassischen, seriellen Vorgehensweisen wie zum Beispiel dem Wasserfall-Modell vor, dass sämtliche Testfälle einer Funktionalität bereits *vor* dessen Umsetzung geschrieben werden müssen [15]. Die Vorteile dieser Methodik ist die Gewährleistung einer hohen Testabdeckung [16, S. 90] und die Erzielung einer Implementierung, welche die Anforderungen vollständig erfüllt, sofern die Testfälle sorgfältig konstruiert werden [16, S. 214]. Wenn die Testfälle erst nach der Programmierung angelegt werden, besteht die Gefahr, dass diese lediglich die tatsächlich umgesetzten Funktionen betrachten, jedoch nicht den ursprünglichen, möglicherweise abweichenden Anforderungen gerecht werden.

Der Testaufbau wurde im konkreten Fall wie folgt konzipiert: Pro Modul wird ein Verzeichnis mit einer Eingabe- und einer Ausgabedatei angelegt. Die Eingabe enthält dabei reguläres JavaScript, das mit Flow typisiert wurde, die Ausgabe den äquivalenten, manuell übersetzten TypeScript-Code. In den Dateien können beliebig viele Testfälle einer Kategorie spezifiziert werden, um eine bestimmte Funktionalität des Transpilers zu erproben. Derartige Dateien oder

[19] Vgl. Abschnitt 3.3.1.
[20] Engl. *test-driven development* (TDD).

Objekte, die der Initialisierung von Modultests dienen, werden oft als „*Fixtures*“ bezeichnet [103]. Durch die bewusste Aufteilung auf zwei unabhängige Dateien kann die inhärente Validität der jeweiligen Quelltexte besser gewährleistet werden, da diese jeweils mittels Flow bzw. TypeScript auf Fehler überprüft werden können. Hierdurch wird vermieden, dass bereits die Testfälle fehlerhaft hinsichtlich der Syntax oder der Typisierung erstellt werden. Würden die Quelltexte als Zeichenketten innerhalb der Testumgebung angegeben werden, so wäre eine derartige Prüfung durch das statische Typsystem nicht möglich.

Die Testdurchführung kann nun wie folgt umgesetzt werden: Der Transpiler wird auf die Eingabedatei angewandt und der auf diese Weise generierte TypeScript-Code wird anschließend Zeile für Zeile mit der erwarteten Ausgabe exakt verglichen. Um dies zu erreichen, wurde ein Skript geschrieben, welches das Verzeichnis mit den Modultests einliest, die Transpilierung anstößt und anschließend den zeilenweisen Vergleich durch das Test-Framework *Jest* [53] durchführt. Das angegebene Verzeichnis kann dabei, wie in Quelltext 4.1 veranschaulicht, beliebig tief verschachtelte Unterverzeichnisse mit Fixture-Dateien enthalten.

```
tests/fixtures
└── types
    ├── any
    │   ├── input.js
    │   └── output.ts
    ├── ...
    └── utility
        └── call
            ├── input.js
            └── output.ts
```

Quelltext 4.1: Fixture-Dateien zum Test der korrekten Transpilierung der Flow-Typen.

Während der Entwicklung des Transpilers wurden nach und nach Modultests für alle Funktionen des Programms angelegt. Überprüft wird dabei die korrekte Übersetzung aller Flow-Typen, die Richtigkeit weiterer Optimierungen und die Formatierung der Ausgabe. Weiterhin wurden Tests zur Erprobung des Kommandozeilenprogramms und der allgemeinen Hilfsfunktionen geschrieben. Damit Regression während der fortlaufenden Entwicklung unmittelbar festgestellt werden, werden die Tests nach Einspielung von Änderungen in das Versionverwaltungssystem automatisch ausgeführt (*kontinuierliche Integration*). Insgesamt sind 1022 Testfälle[21] entstanden und es wurde eine Testabdeckung von 93% erreicht.

[21] Anmerkung: Diese hohe Zahl ist darin begründet, dass größtenteils Fixture-Tests umgesetzt wurden und jeder Vergleich einer nicht-leeren Zeile hierbei als ein Test betrachtet wird.

4.2.2 Statische Typisierung von Babel

Der gesamte Transpiler, also sowohl das Babel-Plugin, als auch das zugehörige Kommandozeilenprogramm, wurde in TypeScript implementiert. Neben den allgemeinen, bereits ausgeführten Vorteilen statischer Typsysteme ist der primäre Grund hierfür, dass so die durch Babel bereitgestellten Typdeklarationen benutzt werden können. Diese unterstützen den Entwicklungsprozess sehr, weil die Attribute aller Knoten des abstrakten Syntaxbaums von Babel und die Signatur sämtlicher Methoden durch die Typisierung genau spezifiziert sind. Somit kann während der Entwicklung durch den TypeScript-Compiler unmittelbar festgestellt werden, ob fehlerhafte Aufrufe von Bibliotheksfunktionen oder anderweitige Typverletzungen bestehen. Folglich werden inkorrekte Transformationen und Laufzeitfehler reduziert.

Quelltext 4.2 zeigt exemplarisch den Aufbau einer solchen Typdeklaration von Babel anhand des Knotens für ein Typalias in Flow. Dieses besteht aus einem Bezeichner (`id`) und einem zugewiesenen Flow-Typ (`right`). Der zugewiesene Typ muss dabei ein Element des Vereinigungstyp `FlowType` sein, der alle möglichen Typen von Flow umfasst. Darüber hinaus können optional Typparameter des Alias angegeben werden.

```
1  interface TypeAlias extends BaseNode {
2    type: "TypeAlias";
3    id: Identifier;
4    typeParameters: TypeParameterDeclaration | null;
5    right: FlowType;
6  }
```

Quelltext 4.2: Externe Typdeklaration des AST-Knotens *TypeAlias* in Babel (Flow).

Durch Vergleich der abstrakten Syntaxbäume eines äquivalenten Programmtexts in Flow und TypeScript ist nachvollziehbar, welche TypeScript-Token einem bestimmten Flow-Ausdruck entsprechen. Weil alle Methoden von Babel typisiert sind, ist damit auch bekannt, welche Typen als Argument der Funktion, die den bedeutungsgleichen TypeScript-Ausdruck herstellt, erwartet werden. Die Implementierung von Transformationen wird hierdurch erheblich vereinfacht, da so lediglich die Attribute des ursprünglichen Knotens richtig auf den Typ des jeweiligen Funktionsparameters abgebildet werden müssen. Es hat sich während der Entwicklung gezeigt, dass eine derartige Einschränkung durch die Typisierung sehr hilfreich ist: Wenn keine Typfehler mehr vorliegen, ist die Transformation des Flow-Typs nach TypeScript erfahrungsgemäß mit hoher Wahrscheinlichkeit korrekt.

4.3 Implementierung als Babel-Plugin

Nachfolgend soll nun der Aufbau und die konkrete Umsetzung des Babel-Plugins, das die Übersetzung der Flow-Typen nach TypeScript realisiert, erläutert werden.

4.3.1 Überblick über Ablauf der Transpilierung

Abbildung 4.2 auf Seite 43 veranschaulicht den konzeptionellen Aufbau der Transpilierung innerhalb des Babel-Plugins anhand eines Aktivitätsdiagramms. Aktivitätsdiagramme entstammen der Modellierungssprache *Unified Modeling Language* (UML) [102] und veranschaulichen die Arbeitsweise von Prozessen innerhalb eines Softwaresystems. Zu Beginn wird das Plugin wie im Grundlagenteil in Quelltext 2.4 bereits exemplarisch gezeigt initialisiert, das heißt, es wird eine Abbildung der Flow-Knotentypen des abstrakten Syntaxbaums auf Besucherfunktionen definiert. Diese realisieren die Transformation aller Flow-Typannotationen in entsprechende TypeScript-Syntax. Weiterhin werden die Abhängigkeiten des Plugins spezifiziert. Dabei handelt es sich um weitere vorgegebene Babel-Plugins, die benötigt werden, um die Syntax von Flow, JSX und vorläufiger ECMAScript-Erweiterungen einlesen zu können[22]. Auf Grundlage der konkreten Anforderungen bei TeamShirts wurden externe Plugins für folgende experimentellen Erweiterungen aktiviert:

- *Class field declarations for JavaScript* [45]
- *JavaScript decorators* [44]
- *Dynamic imports* [35]

Nach der Initialisierung beginnt die rekursive Traversierung des abstrakten Syntaxbaums der Eingabe, um diese umzuformen. Die Datenstruktur des Syntaxbaums wird von Babel durch Parsen des Flow-Programms aufgebaut. Jeder Knoten des Baums wird dabei zunächst „betreten“ und daraufhin wieder „verlassen“ [89]. Knoten werden verlassen, wenn bei der Beschreitung eines Pfads ein Blatt des Baums erreicht wird und die Traversierung daraufhin auf der höherliegenden Ebene fortgesetzt wird.

[22] Vgl. Anforderung 3.3.3.

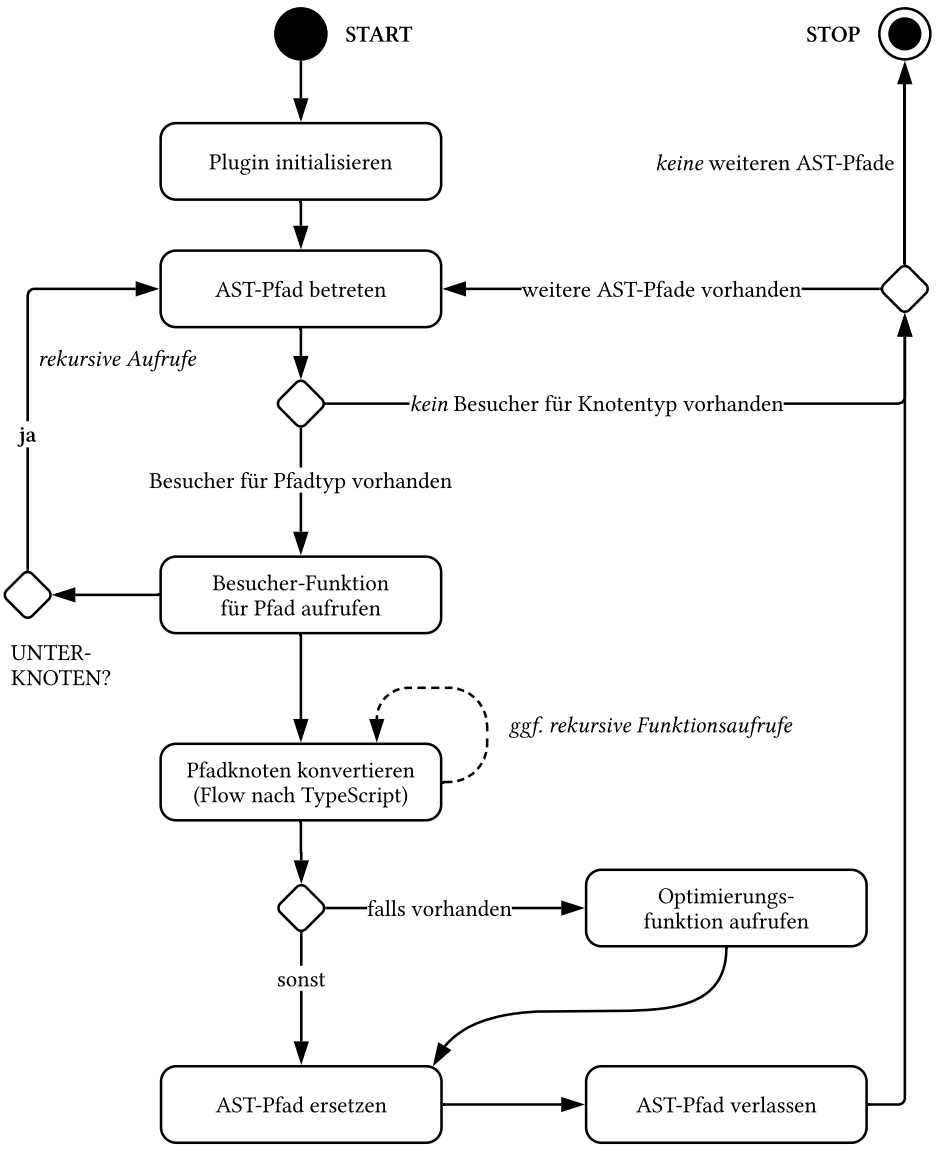


Abbildung 4.2: Aktivitätsdiagramm des Transpilers (Babel-Plugin).

Sofern eine Besucherfunktion für den aktuellen Knotentyp existiert, wird diese mit dem zugehörigen Pfad als Argument aufgerufen. Ansonsten wird das nächste Element des Syntaxbaums betreten bzw. die Transpilierung beendet, falls keine weiteren Knoten bestehen. Wenn innerhalb der Besucherfunktion festgestellt wird, dass Kindknoten vorliegen, so wird deren Pfad rekursiv beschritten. Andernfalls wird der Konverter für diesen speziellen Knotentyp ausgeführt, sodass der momentane Pfad anschließend mit dem auf diese Weise berechneten TypeScript-Gegenstück ersetzt werden kann. Gleichzeitig werden für einige Knotenarten Optimierungsfunktionen aufgerufen. Diese korrigieren beispielsweise fehlerhaft verwendete Typen und sind auf die Gegebenheiten der konkreten JavaScript-Projekte abgestimmt. Da viele der Typannotationen von Flow durch mehrere Typen zusammengesetzt werden (zum Beispiel *Union type*, *Intersection type*, usw.), werden diese Ausdrücke durch mehrere rekursive Aufrufe von Konvertern umgeformt. Zuletzt wird der momentane Pfad verlassen und die Prozedur beginnt von neuem, sofern weitere Elemente im abstrakten Syntaxbaum vorliegen. Sobald das gesamte Programm und damit sämtliche Flow-Typen übersetzt wurden, kann die TypeScript-Ausgabe mittels des Codegenerators von Babel [11] erzeugt werden.

4.3.2 Transpilierung der Flow-Typen

Im Folgenden soll nun genauer auf den Kern des Transpilers, die Konverterfunktionen, eingegangen werden. Diese realisieren die Übersetzung der einzelnen Flow-Typen nach TypeScript. Aufgrund der großen Zahl von Typen kann nicht die vollständige Umsetzung aller Transformationen und sämtlicher Grenzfälle ausführlich dargelegt werden, da dies den Umfang der Arbeit überschreiten würde. Deshalb wird nachfolgend lediglich ein Überblick über alle Übersetzungen gegeben. Jedoch wurden zur Verbesserung der Anschaulichkeit darüber hinaus einige repräsentative Beispiele ausgewählt, anhand derer das Prinzip der Transpilierung detaillierter erläutert wird, indem die zugrunde liegende Implementierung betrachtet wird. Die genaue Umsetzung aller Transformationen und die Behandlung der Spezialfälle kann mittels des veröffentlichten Quelltexts des Transpilers [71] nachvollzogen werden.

Transpilierung der Basistypen

Die Mehrheit der in Tabelle 2.1 vorgestellten Basistypen von Flow kann innerhalb des Plugins simpel übersetzt werden, da der entsprechende Typ in TypeScript die gleiche Syntax besitzt

oder sich lediglich das Schlüsselwort unterscheidet. Von den insgesamt 30 Basistypen können 18 dieser einfachen Kategorie zugeordnet werden. Die verbleibenden zwölf Typen erfordern komplexere Knotentransformationen, um die Annotationen in entsprechende TypeScript-Ausdrücke umzuwandeln.

Simple Übersetzungen Tabelle 4.1 listet die 18 Flow-Typen auf, die simpel übersetzt werden können und zeigt beispielhaft deren Entsprechung in TypeScript. Abgesehen von den zwei kursiv hervorgehobenen Zeilen ist die Syntax in TypeScript identisch mit der ursprünglichen Notation der Typen in Flow. Dennoch müssen auch diese Elemente des abstrakten Syntaxbaums während der Transpilierung in ihr korrektes Gegenstück umgewandelt werden, da eine Kombination von Flow- und TypeScript-Knoten innerhalb des abstrakten Syntaxbaums gemäß der Spezifikation [92] von Babel verboten ist. Diese Einschränkung wird zur Laufzeit durch Babel mit Hilfe der Bibliothek `@babel/types` [13] sichergestellt. Aufgrund der Verwendung von TypeScript und der gegebenen Typisierung von Babel werden Typfehler dieser Art aber ohnehin bereits vor Ausführung des Transpilers statisch erkannt.

Tabelle 4.1: Übersicht über simple Transformationen der Basistypen von Flow.

Basistyp	**Flow**	**TypeScript**
Any type	`any`	`any`
Array type	`Array<number>` `number[]`	`Array<number>` `number[]`
Class type	`class C {}` `type ClassAlias = C`	`class C {}` `type ClassAlias = C`
Boolean literal type	`true`	`true`
Boolean type	`boolean`	`boolean`
Empty type	`empty`	`never`
Intersection type	`type Intersection = A & B`	`type Intersection = A & B`
Mixed type	`mixed`	`unknown`
Null literal type	`null`	`null`
Number literal type	`42`	`42`
Number type	`number`	`number`
String literal type	`'literal'`	`'literal'`
String type	`string`	`string`
This type	`this`	`this`
Tuple type	`[Date, number]`	`[Date, number]`
Type alias	`type Type = <FlowType>`	`type Type = <TSType>`
Union type	`number | string | null`	`number | string | null`
Void type	`void`	`void`

```
1  // @flow                          // TypeScript
2  type Alias = mixed;               type Alias = unknown;
```

Quelltext 4.3: Beispiel für die Übersetzung simpler Flow-Typen.

Nachfolgend soll das grundsätzliche Vorgehen bei der Implementierung der Typtransformationen anhand des Ablaufs einer solchen einfachen Übersetzung erläutert werden. Quelltext 4.3 zeigt links das mit Flow typisierte Ursprungsprogramm und rechts den äquivalenten TypeScript-Code. In der zweiten Zeile wird dabei ein Typalias definiert, welcher auf den Typ `mixed` verweist. Wie Tabelle 4.1 angibt, ist der korrespondierende TypeScript-Typ `unknown`. Um die notwendigen Schritte einer entsprechenden Transformation zu erkennen, ist es hilfreich, die abstrakten Syntaxbäume des ursprünglichen und des angestrebten Quelltexts zu vergleichen. Die jeweiligen Syntaxbäume werden in Abbildung 4.3 für Flow (links) und TypeScript (rechts) dargestellt[23]. In den Knoten des Baums steht dabei jeweils der Typ des eingelesenen Tokens, die Kanten repräsentieren dessen Attribute. Konkrete Werte wie beispielsweise der Name von Bezeichnern werden in Anführungszeichen abgedruckt.

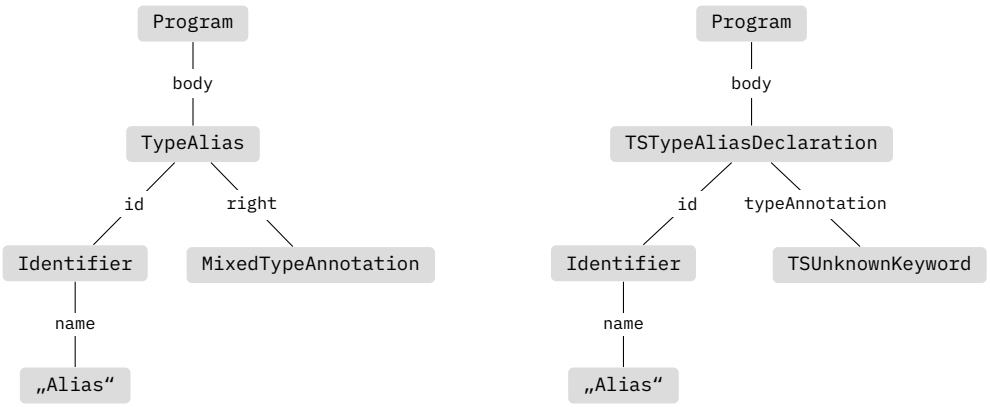


Abbildung 4.3: Abstrakter Syntaxbaum für die Deklaration eines Typalias in Flow (links) und TypeScript (rechts).

Die zwei Bäume sind sich sehr ähnlich: In der Wurzel der Datenstruktur befindet sich bei allen durch Babel eingelesenen Quelltexten der Knoten `Program`. Dessen Kindknoten repräsentiert die Deklaration des Typalias aus der zweiten Zeile des Beispiels. Der Typ dieses AST-Elements ist bei Flow `TypeAlias` und bei TypeScript `TSTypeAliasDeclaration`. Auf der linken Seite der Anweisung steht dabei der Bezeichner des Alias und auf der rechten Seite der zugewiesene

[23] Anmerkung: Zur Vereinfachung der Nachvollziehbarkeit wurde der Syntaxbaum auf seine wesentlichen Teile reduziert. Tatsächlich besitzen alle Knoten weitere Attribute, die hier jedoch irrelevant sind.

Typ. Weil im ursprünglichen Code das Alias auf `mixed` (`MixedTypeAnnotation`) verweist, muss dieser Knoten in TypeScript zu `unknown` (`TSUnknownKeyword`) übersetzt werden, um einen äquivalenten Ausdruck zu erzeugen. In diesem konkreten Fall sind die zwei notwendigen Transformationschritte damit folgende:

- Übersetzung des `TypeAlias`-Knotens nach `TSTypeAliasDeclaration`
- Übersetzung des `MixedTypeAnnotation`-Knotens nach `TSUnknownKeyword`

Der erste Schritt, also die Übersetzung des Typalias, wird durch eine Konverterfunktion realisiert, die innerhalb des Besuchers für `TypeAlias`-Knoten aufgerufen wird. Quelltext 4.4 zeigt deren vereinfachten Aufbau. Hierbei werden in Zeile 4 zunächst die Typparameter durch Aufruf einer weiteren Konvertierungsfunktion in das entsprechende TypeScript-Element transformiert. Daraufhin wird der zugewiesene Typ des Alias übersetzt (Zeile 5). Weil dabei ein *beliebiger* Flow-Typ möglich ist, muss dieser allgemeingültig umgewandelt werden können. Es wurde deshalb eine Funktion `convertFlowType` implementiert, die alle Flow-Typen universell auf ihr TypeScript-Gegenstück abbildet.

```
1  function convertTypeAlias(
2    node: TypeAlias
3  ): TSTypeAliasDeclaration | TSInterfaceDeclaration {
4    const tsTypeParameters = convertTypeParameterDeclaration(node.typeParameters);
5    const tsTypeAnnotation = convertFlowType(node.right);
6
7    if (isInterfaceTypeAnnotation(node.right))
8      return convertInterfaceTypeAlias(node);
9
10   return tsTypeAliasDeclaration(node.id, tsTypeParameters, tsTypeAnnotation);
11 }
```

Quelltext 4.4: Konvertierungsfunktion für Typaliase.

Babel stellt für sämtliche Knoten des abstrakten Syntaxbaums Methoden bereit, mittels derer diese erzeugt werden können [13]. Somit kann in Zeile 10 durch Aufruf der Bibliotheksfunktion `tsTypeAliasDeclaration` ein TypeScript-Typalias erstellt werden, indem der gleiche Bezeichner sowie die übersetzten Typparameter und der transformierte zugewiesene Typ angegeben werden. In Zeile 7 f. wird außerdem ein Spezialfall behandelt: Während Flow Typaliase für Schnittstellen (*Interfaces*) erlaubt, stellt dies in TypeScript unzulässige Syntax dar. Deshalb werden derartige Typaliase in einen regulären Schnittstellentyp umgeformt.

Der schematische Aufbau der universellen Umwandlungsfunktion `convertFlowType` wird in Quelltext 4.5 gezeigt. In Zeile 10 wird hierbei der zweite Schritt der AST-Transformation, die Übersetzung des `mixed`-Schlüsselworts nach `unknown`, umgesetzt. Durch Aufruf der Bibliotheksfunktion `tsUnknownKeyword()` wird ein neuer AST-Knoten erzeugt, der das entsprechende Schlüsselwort in TypeScript repräsentiert. In Zeile 4 wird ein weiteres wichtiges Konzept innerhalb des Transpilers exemplarisch aufgezeigt: Für Knoten des Typs `ArrayTypeAnnotation` wird die Umwandlungsfunktion *rekursiv* aufgerufen, da auch hier beliebige Flow-Typen als Argument des Feldtyps angegeben werden können. Wie die Transformation des Typalias bereits zeigt, spielt die Umwandlungsmethode `convertFlowType` eine zentrale Rolle innerhalb des Transpilers, da sie in nahezu allen Konvertern aufgerufen wird. Zeile 7 verdeutlicht schließlich, dass nicht alle Knoten so einfach wie der Typ `mixed` aus dem Beispiel umgewandelt werden können, da der Aufbau vieler Typen komplexer ist. Die Transformation dieser Knoten wird separat durch jeweilige Konverterfunktionen durchgeführt. Im Folgenden wird die Übersetzung solcher komplizierteren Typen betrachtet.

```
1  function convertFlowType(node: FlowType): TSType {
2    switch (node.type) {
3      case 'ArrayTypeAnnotation':
4        return tsArrayType(convertFlowType(node.elementType));
5      // ...
6      case 'InterfaceTypeAnnotation':
7        return convertInterfaceTypeAnnotation(node);
8      // ...
9      case 'MixedTypeAnnotation':
10       return tsUnknownKeyword();
11   }
12 }
```

Quelltext 4.5: Universelle Transpilierung beliebiger Flow-Typen nach TypeScript durch zentrale Umwandlungsfunktion.

Komplexere Übersetzungen Die Übersetzung der zwölf komplexen Flow-Datentypen wird in Tabelle 4.2 exemplarisch gezeigt. Im Gegensatz zu den simplen Transformationen unterscheidet sich die Syntax des Ausgabequelltexts hier zum Teil deutlich und es müssen bei der Transpilierung vermehrt Spezial- und Grenzfälle beachtet werden, um korrekten TypeScript-Code zu erzeugen. Zum Beispiel erlaubt Flow den Typ einer Funktion, die eine Zeichenkette und eine Zahl als Argument entgegen nimmt, wie folgt zu deklarieren:

```
type FunctionType = (string, argName: number) => void;
```

Während die Benennung der formalen Parameter in Flow optional ist, ist diese in TypeScript verpflichtend. Deshalb müssen Parameter während der Übersetzung von Funktionstypen nach TypeScript gegebenenfalls automatisch benannt werden, da andernfalls fehlerhafte Syntax entstünde[24]. Der in der Tabelle kursiv hervorgehobene Typ *Opaque type* wird in TypeScript nicht unterstützt [118] und wird deshalb, wie gezeigt, zu einem regulären Typalias umgeformt.

Tabelle 4.2: Übersicht über komplexe Transformationen der Basistypen von Flow.

Basistyp	**Flow**	**TypeScript**
Exact object type	`{| prop: boolean |}`	`{ prop: boolean }`
Function type	`(string, arg?: {}) => number`	`(p1: string, arg?: ) => number;`
Generic type annotation	`let v: <FlowType>`	`let v: <TSType>`
Generics	`type Generic<T: Super> = T`	`type Generic<T extends Super> = T`

Interface type	`interface I {`
	`  prop: mixed;`
	`  +covariant: number[];`
	`}`
		`interface I {`
		`  prop: unknown;`
		`  readonly covariant: number[];`
		`}`

Nullable type (Maybe type)	`?number`	`number | null | undefined`

Object type	`{`
	`  [key: string]: number;`
	`  prop: string;`
	`}`
		`{`
		`  [key: string]: number;`
		`  prop: string;`
		`}`

Opaque type	`opaque type O = number`	`type O = number`
Type cast expresssion	`(variable: string)`	`(variable as string)`
Type export	`export type T = number | null`	`export T = number | null`
Type import	`import type T from './types'`	`import T from './types'`
Typeof type	`typeof undefined`	`undefined`

Anhand des Typs *Nullable* soll im Folgenden das Vorgehen bei der Transformation eines komplizierteren Knoten erläutert werden[25]. Dieser Typ wird in der Flow-Dokumentation auch als „*Maybe type*“ bezeichnet und für die Typisierung optionaler, möglicherweise undefinierter Werte verwendet [55]. Konkret stellt der Typ die Vereinigung aus dem angegebenen Typ, `null` und `undefined` dar. In TypeScript existiert keine direkte Entsprechung, aber es kann trivial ein äquivalenter Vereinigungstyps angegeben werden. Quelltext 4.6 zeigt die jeweilige Deklaration eines Typalias in Flow (links) und TypeScript (rechts), das auf diesen Datentyp verweist.

[24] Vgl. Zeile 2 in Tabelle 4.2.

[25] Anmerkung: Es wurde ein verhältnismäßig unkompliziertes Beispiel ausgewählt, weil die Ausführung anderer deutlich komplexerer Fälle nicht in prägnanter Form erfolgen kann.

```
1  // @flow                              // TypeScript
2  type MaybeNumber = ?number;           type MaybeNumber = number | null | undefined;
```

Quelltext 4.6: Beispiel für die Übersetzung komplexer Flow-Typen.

Wie bei der vorherigen Betrachtung einer simplen Übersetzung (`MixedTypeAnnotation`) sollen auch hier zunächst die abstrakten Syntaxbäume der Ein- und Ausgabe für den Typ *Nullable* verglichen werden, um die Einzelschritte der Transpilierung aufzuzeigen (vgl. Abbildung 4.4). Das Typalias ist dabei analog zum vorherigen Beispiel und dient lediglich der Herstellung korrekter Syntax. Entscheidend ist der zugewiesene Typ des Alias im rechten Teilbaum: Bei Flow liegt hier das Element `NullableTypeAnnotation` vor, das den Typ für Zahlen (`NumberTypeAnnotation`) als Argument erhält. Auf Seite von TypeScript steht dagegen der Knoten des Vereinigungstyps (`TSUnionType`). Diesem ist eine Menge von Typen zugeordnet, die den äquivalenten Ausdruck bilden, indem der Zahlentyp mit den Typen für `null` und `undefined` kombiniert wird.

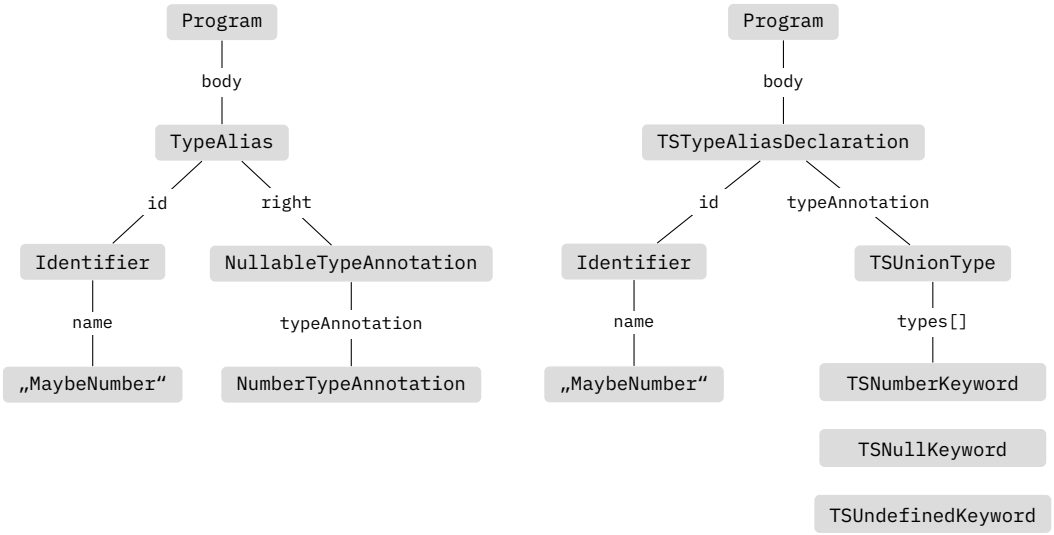


Abbildung 4.4: Abstrakter Syntaxbaum für die Deklaration eines „Maybe type“ in Flow (links) und TypeScript (rechts).

Quelltext 4.7 zeigt die entsprechende Implementierung. Zuerst wird in Zeile 2 das Argument des Typs `NullableTypeAnnotation` durch Aufruf der bereits gezeigten zentralen Umwandlungsfunktion von Flow in den korrekten TypeScript-Typ übersetzt. Im vorliegenden Fall wird Flows `NumberTypeAnnotation` in das Gegenstück `TSNumberKeyword` überführt. Anschließend wird eine Liste aufgebaut, die aus diesem Datentyp und den Typen für `null` und `undefined`

besteht. Zu Beachten sind hierbei zwei Spezialfälle: Einerseits darf der Nullwert nicht doppelt in die Liste aufgenommen werden (Zeile 4 f.), damit kein redundanter Ausdruck entsteht, andererseits müssen Funktionstypen in dieser Situationen geklammert werden, um korrekte TypeScript-Syntax herzustellen (Zeile 6). Eine Doppelung träte dann auf, wenn in Flow der Typ `?null` verwendet wird. Zuletzt kann der TypeScript-Vereinigungstyp durch Aufruf der entsprechenden Bibliotheksfunktion von Babel mit der Typliste als Argument erzeugt und zurückgeliefert werden.

```
1  function convertNullableTypeAnnotation(node: NullableTypeAnnotation): TSUnionType {
2    const tsType = convertFlowType(node.typeAnnotation);
3    const types = [
4      ...(isTSNullKeyword(tsType)
5        ? []
6        : [isTSFunctionType(tsType) ? tsParenthesizedType(tsType) : tsType]),
7      tsNullKeyword(),
8      tsUndefinedKeyword(),
9    ];
10
11   return tsUnionType(types);
12 }
```

Quelltext 4.7: Übersetzung eines *Maybe types* in äquivalenten Vereinigungstyp in TypeScript.

Transpilierung der Hilfstypen

Auch die in Abschnitt 2.2.1 eingeführten Hilfstypen von Flow können größtenteils äquivalent nach TypeScript überführt werden. Tabelle 4.3 gibt einen Überblick über alle Übersetzungen. Für einige der Einträge gibt es in TypeScript bedeutungsgleiche Hilfstypen, andere können leicht durch Verwendung von Typoperatoren abgebildet werden.

Tabelle 4.3: Übersicht über Transformationen der Hilfstypen von Flow.

Hilfstyp	Flow	TypeScript	Anmerkung
Call	`$Call<F, T...>`	`ReturnType<F>`	
Class	`Class<C>`	`typeof T`	
Difference	`$Diff<A, B>`	`Omit<A, keyof B>`	
Element type	`$ElementType<T, K>`	`T[k]`	
Exact	`$Exact<O>`	`O`	
Existential type	`*`	`any`	Verlust von Typinformation!

Übersicht über Transformationen der Hilfstypen von Flow.

Hilfstyp	**Flow**	**TypeScript**	**Anmerkung**
Keys	`$Keys<O>`	`keyof O`	
None maybe type	`$NonMaybeType<T>`	`NonNullable<T>`	
Object map	`$ObjMap<O, F>`	`any`	nicht implementiert
Object map with key	`$ObjMapi<O, F>`	`any`	nicht implementiert
Property type	`$PropertyType<O, k>`	`O[k]`	
Read only	`$ReadOnly<O>`	`ReadOnly<O>`	
Read only array	`$ReadOnlyArray<A>`	`ReadonlyArray<A>`	
Rest	`$Rest<A, B>`	`Omit<A, Union<keyof B>>`	
Shape	`$Shape<O>`	`Partial<O>`	
Tuple map	`$TupleMap<T, F>`	`any`	nicht implementiert
Values	`$Values<O>`	`O[keyof O]`	
Subtype	`$Subtype<T>`	`any`	obsolet
Supertype	`$Supertype<T>`	`any`	obsolet

Die Transformation der drei Abbildungstypen *Object map*, *Object map with key* und *Tuple map* wurde nicht umgesetzt, weil kein TypeScript-Ausdruck gefunden werden konnte, der diesen exakt entspricht. Deshalb werden diese Typen bei der Transpilierung mit *any* ersetzt. Da die JavaScript-Projekte von TeamShirts diese Hilfstypen jedoch ohnehin nicht verwenden, stellt dies kein Hindernis für die Migration im vorliegenden Fall dar. Die zwei Typen *Subtype* und *Supertype* wurden wie ausgeführt von Flow als veraltet (*deprecated*) markiert und werden deshalb ebenfalls nach `any` übersetzt. Auch der *Existential type* wurde als überholt gekennzeichnet, da er unsicher ist [54]. Weil TypeScript einen derartigen Typ nicht unterstützt, wird auch dieser durch `any` ersetzt. Sollten in der Eingabe einer dieser sechs nicht unterstützen Hilfstypen auftreten, so wird während der Verarbeitung durch das Babel-Plugin wie gefordert eine detaillierte Warnung ausgegeben mit der Aufforderung, den Ausgabequelltext an dieser Stelle manuell zu korrigieren[26].

Zu beachten ist darüber hinaus, dass die Typparameter der Hilfstypen (`F`, `T`, `O` usw.) für beliebige kompatible Flow-Typen stehen. Deshalb ruft die Implementierung auch hier intern die universelle Konvertierungsfunktion `convertFlowType()` auf, um alle Fälle korrekt zu übersetzen.

[26] Vgl. Anforderung 3.3.1.

Transpilierung der Typdeklarationen

Da der Transpiler Anspruch auf Vollständigkeit erhebt, müssen auch Typdeklarationen transformiert werden. Tabelle 4.4 zeigt exemplarisch die verschiedenen Deklarationen und deren Gegenstücke in TypeScript. Die Syntax ist hierbei ähnlich zu den simplen Basistypen oftmals identisch in Flow und TypeScript. Dennoch müssen die zugrunde liegenden Knoten des abstrakten Syntaxbaums übersetzt werden, da die entsprechenden Ausdrücke in TypeScript eigene Knotentypen besitzen. Interessant ist die Deklaration des Standardexports eines Moduls[27] in der zweiten Zeile. Während Flow hier den Export von Funktionen unterstützt, kann dies nicht unmittelbar in TypeScript abgebildet werden. Deshalb muss die Deklaration umgeformt werden: Der Wert der ursprünglichen Deklaration wird dabei einer neu erstellten Variablen (`_default`) zugewiesen und kann anschließend als Argument des Exports verwendet werden, sodass ein äquivalenter Ausdruck entsteht.

Tabelle 4.4: Übersicht über Transformationen der Typdeklarationen von Flow.

Deklaration	**Flow**	**TypeScript**
Class	`declare class C {}`	`declare class C {}`
Export	`declare export default () => string`	`const _default: () => string;` `export default _default;`
Function	`declare function f(number): any`	`declare function f(p: number): any`
Interface	`declare interface I {}`	`declare interface I {}`
Module	`declare module 'M' {}`	`declare module 'M' {}`
Type alias	`declare type T = number`	`declare type T = number`
Variable	`declare var v: ?string`	`declare var v: string | null | undefined`

4.3.3 Weitere Optimierungen

Bei der Erprobung des Transpilers mit den zwei JavaScript-Projekten von TeamShirts wurde die Erfahrung gewonnen, dass neben der Transformation der Flow-Typen auch andere Änderungen des Quelltexts nötig sind, um eine Migration zu TypeScript erfolgreich durchzuführen. Zur Reduzierung händischer Nacharbeit wurde das Plugin um Funktionen erweitert, die diese zusätzlichen Schritte automatisieren. Diese werden nachfolgend ausgeführt.

[27] Siehe [42, S. 377].

Übersetzung von React-Typimporten

Die von TeamShirts eingesetzte Bibliothek React stellt ähnlich wie Babel Typdeklarationen sowohl für Flow als auch für TypeScript bereit. Diese Typen können importiert und anschließend in den eigenen Typannotation verwendet werden. Leider wurden die korrespondierenden React-Typen in den Definitionen für Flow und TypeScript unterschiedlich benannt. Während einer dieser Typen in Flow beispielsweise `Node` heißt, ist dessen Bezeichnung in TypeScript `ReactNode`. Würden die Typimporte unverändert übernommen werden, so entstünden in der Ausgabe Typfehler, da der importierte Typ unter TypeScript nicht existiert.

```
1 // @flow                                    // TypeScript
2 import React, { type Node } from 'react';   import React, { ReactNode } from 'react';
3 type FunctionType = () => Node;             type FunctionType = () => ReactNode;
```

Quelltext 4.8: Import und Benutzung des durch die Bibliothek *React* extern definierten Typs *Node*.

Quelltext 4.8 veranschaulicht, wie die Transpilierung richtig durchgeführt werden sollte. Zum einen muss die Syntax des Typimports an sich transformiert werden, zum anderen sollten die Namen sämtlicher React-Typen korrekt auf ihr Gegenstück in TypeScript übersetzt werden. Dabei müssen auch alle Konstrukte, die diese importierten Typen verwenden, entsprechend angepasst werden. Da bekannt ist, welche Typen von React exportiert werden und wie deren Bezeichnung in TypeScript ist, konnte eine solche Transformation mittels einer statischen Abbildungstabelle implementiert werden.

Konvertierung von Klassendekoratoren

In den Projekten von TeamShirts werden Klassendekoratoren verwendet, um React-Komponenten um verschiedene Funktionen zu erweitern. Derartige Dekoratoren sind eine vorgeschlagene Spracherweiterung von ECMAScript, die es ermöglichen, Klassen oder deren Attribute unabhängig von der zugrunde liegenden Implementierung zu modifizieren [44]. Dabei werden spezielle Annotationen (`@decorator`) in den Quelltext unmittelbar vor der Deklaration einer Klasse eingefügt. Dekoratoren stellen Funktionen höherer Ordnung dar, die den Konstruktor der Klasse als Argument erhalten, diesen gegebenenfalls verändern und daraufhin eine Funktion zurückliefern, die den Konstruktor intern aufruft [44]. Auf diese Weise kann die Instanziierung aller Objekten dieser Klasse beeinflusst werden, indem beispielsweise weitere Methoden vor oder nach Aufruf des Konstruktors ausgeführt werden.

Es wurde bei der praktischen Erprobung des Transpilers festgestellt, dass die Verwendung von Dekoratoren in TypeScript problematisch ist, da dies zu zahlreichen Typfehlern führt und eine korrekte Typisierung aufwändig ist. Deshalb wurde eine optionale Funktion in den Transpiler integriert, welche die Klassendekoratoren durch äquivalente, verschachtelte Funktionsaufrufe ersetzt. Es hat sich gezeigt, dass diese hinsichtlich der Typisierung weitaus weniger Schwierigkeiten verursachen. In Quelltext 4.9 wird eine derartige Transpilierung veranschaulicht. Die Transformation der Klassendekoratoren kann durch Setzen der Option „`replace-decorators`“[28] des Kommandozeilenprogramms aktiviert werden.

```
// @flow
@d1(arg)
@d2
class C {}
export default C;
```

```
// TypeScript


class C {}
export default d1(arg)(d2(C));
```

Quelltext 4.9: Optionale Übersetzung von Klassendekoratoren in verschachtelte Funktionsaufrufe.

4.4 Erweiterung als Kommandozeilenprogramm

Zur Erfüllung der in Abschnitt 3.3.4 dargelegten Anforderung, dass der Transpiler in der Lage sein muss, gesamte Projektverzeichnisse zu verarbeiten (Stapelverarbeitung), ist die Erweiterung des Transcompilers um ein Kommandozeilenprogramm nützlich. Hierdurch können beliebige Dateien und Verzeichnisse eingelesen und deren Übersetzung durch verschiedene Optionen beeinflusst werden. Wie bereits in Abschnitt 4.1 umrissen, benutzt die Konsolenanwendung intern das Babel-Plugin, um die Transpilierung der Flow-Quelltexte nach TypeScript durchzuführen. Die Aufgaben des Programms sind damit das Einlesen der Eingabe, die Delegation dieser an das Babel-Plugin, die Formatierung des generierten TypeScript-Codes und schließlich die Ausgabe desselben.

Die Anwendung wurde als ausführbares Node.js-Skript umgesetzt und *Reflow* genannt. Das Werkzeug kann durch das Paketsystem von Node.js installiert und anschließend wie folgt aufgerufen werden[29]. Tabelle 4.5 zeigt alle Optionen des Kommandozeilenprogramms und beschreibt deren Zweck.

[28] Vgl. Tabelle 4.5.
[29] Siehe [71] für detaillierte Installations-Anweisungen.

```
reflow [OPTIONEN...] <DATEIEN ODER ...VERZEICHNISSE>
Beispiel: reflow --dry-run --include-pattern "**/*.js" src/
```

Tabelle 4.5: Optionen des Kommandozeilenprogramms (*Reflow*).

Option	**Beschreibung**
`-V --version`	Versionsnummer anzeigen.
`-d --dry-run`	Generierten TypeScript-Code auf Standardausgabe statt in Dateien schreiben (Testlauf).
`-e --exclude-dirs <pattern ...>`	Kommaseparierte Liste von Verzeichnissen, die von der Transpilierung rekursiv ausgeschlossen werden sollen.
`-i --include-pattern <pattern>`	Wildcard-Muster (*glob pattern*) für Eingabedateien bei Angabe von Verzeichnissen (Standardwert: `"**/*.js,jsx"`).
`-r --replace`	Originaldateien (Flow) mit generierten TypeScript-Dateien ersetzen, statt diese beizubehalten.
`-D --replace-decorators`	Klassendekoratoren durch verschachtelte Funktionsaufrufe ersetzen (vgl. Abschnitt 4.3.3).
`-h --help`	Hilfe anzeigen.

Abbildung 4.5 auf Seite 57 zeigt das Aktivitätsdiagramm des Kommandozeilenprogramms. Als Erstes werden dessen Argumente eingelesen und validiert. Dabei wird überprüft, ob alle Eingabeverzeichnisse bzw. -dateien existieren und ob der Dateityp korrekt ist. Erlaubt sind lediglich JavaScript- und JSX-Dateien. Sofern diese Validierung fehlschlägt, wird die Anwendung mit einer Fehlermeldung beendet. Andernfalls wird daraufhin eine Liste der zu übersetzenden Flow-Dateien erstellt. Falls Verzeichnisse als Argument angegeben worden sind, so werden alle Dateien innerhalb dieser, die dem Wildcard-Muster für Eingabedateien (Option „`include-pattern`“) entsprechen, der Liste hinzugefügt. Anschließend wird eine Schleife betreten, die alle auf diese Weise ermittelten Dateien nach und nach verarbeitet. Da invalide Wildcard-Muster durch Benutzereingabe möglich sind, wird erneut überprüft, ob die aktuelle Eingabedatei zulässig ist. Sollte dies nicht der Fall sein, so wird diese übersprungen. Ansonsten wird der Quelltext der Datei durch den Parser von Babel eingelesen und so die Datenstruktur des abstrakten Syntaxbaum des Programms aufgebaut. Daraufhin wird die Transpilierung der Flow-Typisierung durch das Babel-Plugin angestoßen. Auch hier kann es zu Laufzeitfehlern kommen, falls beispielsweise Syntaxfehler innerhalb des Eingabequelltexts vorliegen. In diesem Fall wird die aktuelle Datei ebenfalls übersprungen und eine Warnung ausgegeben.

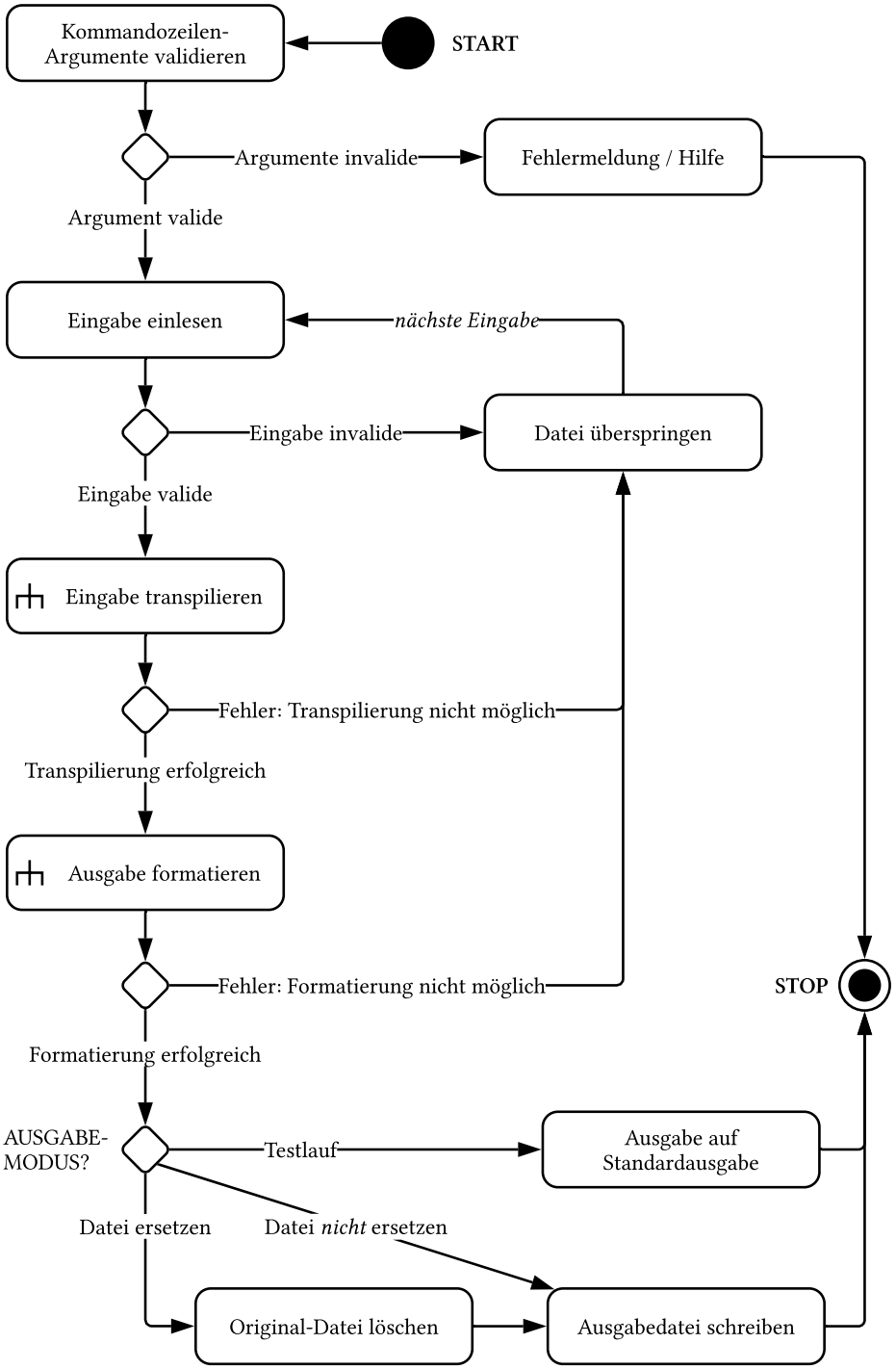


Abbildung 4.5: Aktivitätsdiagramm des Kommandozeilenprogramms. Vgl. eingebettete Diagramme 4.2 „Eingabe transpilieren“ und 4.6 „Ausgabe formatieren“.

Nachdem die Transpilierung abgeschlossen ist, wird der generierte TypeScript-Code unabhängig von Babel formatiert. Dabei wird eine selbst implementierte Funktion ausgeführt, die eine möglichst originalgetreuen Formatierung der Ausgabe umsetzt. Sollten unerwartete Fehler auftreten wird auch hier die Datei gegebenenfalls übersprungen. Zuletzt kann der so erzeugte TypeScript-Quelltext ausgegeben werden. Dabei gibt es je nach Verwendung der Kommandozeilenoptionen drei Möglichkeiten: Wenn der Parameter „`dry-run`“ gesetzt ist, wird das Resultat auf die Standardausgabe geschrieben. Andernfalls werden neue TypeScript-Dateien im Verzeichnis der Eingabedateien erstellt und die Originaldateien anschließend gelöscht, sofern die Option zum Ersetzen dieser angegeben wurde.

Da die Verwendung von JSX-Syntax in TypeScript die Dateierweiterung `.tsx` vorschreibt [84], muss sichergestellt werden, dass die Ausgabedateien die jeweils korrekte Endung erhalten. Weiterhin müssen globale Typdeklarationen in Dateien mit der Erweiterung `.d.ts` geschrieben werden. Zur Bestimmung des richtigen Ausgabedateityps wird deshalb während der Transpilierung innerhalb des Babel-Plugins überprüft, ob JSX-Syntax in der Eingabedatei verwendet wird bzw. ob Typdeklarationen vorliegen. Hierfür werden Besucherfunktionen für die jeweiligen speziellen AST-Knoten registriert, sodass eine Lookup-Tabelle aufgebaut werden kann, welche die Abbildung der Originaldateien auf den korrekten Ausgabetyp darstellt. Das Kommandozeilenprogramm fragt diese Information schließlich beim Schreiben der TypeScript-Dateien ab und reagiert entsprechend.

4.5 Formatierung des Ausgabequelltexts

Ein Aspekt, der sich während der Entwicklung des Transpilers als problematisch herausgestellt hat und deshalb einer gesonderten Betrachtung bedarf, ist die Formatierung des generierten Ausgabecodes. Weil Babel auf Grundlage eines *abstrakten* Syntaxbaums arbeitet, liegt nach der Transformation des Programms keinerlei Information mehr über die ursprüngliche Formatierung des Codes vor. Infolgedessen gehen die Einrückung und die Position der Leerzeichen in der Ausgabe verloren. Auch die Platzierung der Kommentare wird nach der Übersetzung nicht präzise beibehalten. Diese werden deshalb von der Übersetzung ausgeschlossen, weil sie ohnehin teilweise falsch übernommen werden.

Versuche mit der Option „`retainLines`“ [11] des Babel-Codegenerators, welche eine Beibehaltung der Zeilen in der Ausgabe bewirken soll, erzielten leider nicht das gewünschte Ergebnis. Auch nach Setzen dieser Eigenschaft unterscheidet sich das Format des generierten Quelltexts erheblich von der Eingabe. Da die möglichst originalgetreue Formatierung der Ausgabe aber eine der Anforderungen an den Transpiler ist, wurde eine entsprechende Funktion implementiert, um diese Vorgabe unabhängig von Babel zu erfüllen. Das Verfahren wird nach der Transpilierung aller Eingabedateien durch das Kommandozeilenprogramm angestoßen[30] und soll im Folgenden erläutert werden.

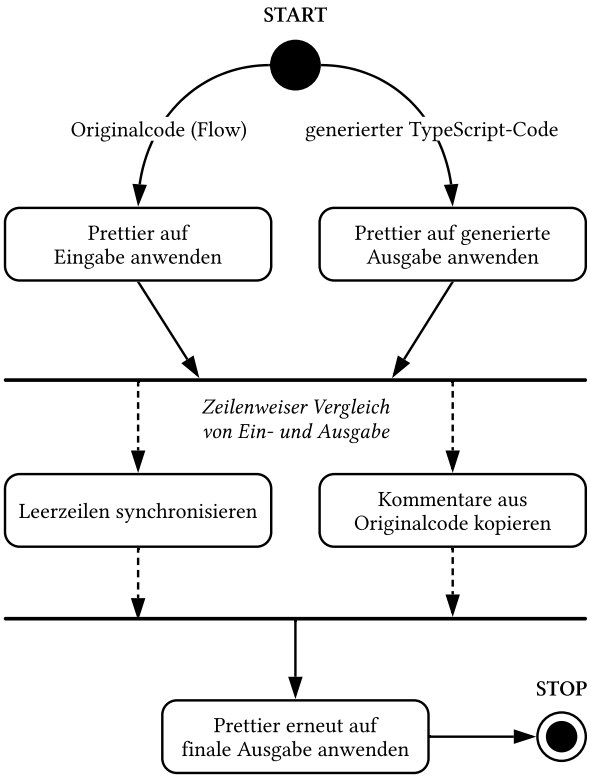


Abbildung 4.6: Aktivitätsdiagramm der Formatierung des generierten Ausgabecodes auf Grundlage des Quelltext-Formatierers *Prettier* [38].

Der konzeptionellen Aufbau der Formatierungsfunktion wird in Abbildung 4.6 dargestellt. Die Idee ist simpel: Nachdem sowohl die Ein- als auch die Ausgabe in einen vergleichbaren, konsistenten Zustand überführt worden sind, können Leerzeilen und Kommentare aus dem

[30] Vgl. Abb. 4.1.

ursprünglichen Quelltext Schritt für Schritt übertragen werden. Konsistent bedeutet, dass die Ausdrücke und Anweisungen in beiden Quelltexten die gleiche Zahl von Zeilen vereinnahmen, also dass diese gleich umgebrochen werden. Diese Voraussetzung muss insbesondere auch für alle Flow-Typen nach deren Übersetzung nach TypeScript gelten. Grund für die Herstellung dieser Eigenschaft ist, dass das weitere Verfahren zeilenbasiert arbeitet, um die Originalformatierung in die Ausgabe zu übernehmen.

Zur Herstellung eines solchen Ausgangszustand wird *Prettier* [38] eingesetzt. Prettier ist ein Quelltext-Formatierer, der einen einheitlichen Programmierstil für Sprachen wie JavaScript, TypeScript, HTML und weitere ermöglicht. Experimente haben gezeigt, dass die Ein- und Ausgabe bei der Anwendung von Prettier allerdings aufgrund unterschiedlicher Syntax von Flow und TypeScript nicht in allen Situationen übereinstimmend umgebrochen wird. Deshalb wurde das Werkzeug geringfügig modifiziert, um so eine größere Konsistenz der Ausgabe zu erzielen[31]. Bei Setzen einer neu hinzugefügten Option werden dabei die Attribute von Objekten und JSX-Ausdrücken stets umgebrochen, um deren zeilenweisen Vergleich zu ermöglichen. Prettier bricht diese standardmäßig erst dann um, wenn die eingestellte Druckweite überschritten wird.

Im ersten Schritt der Formatierungsroutine wird diese angepasste Version von Prettier einerseits auf den Originalcode, andererseits auf den generierte TypeScript-Quelltext angewandt (vgl. Abbildung 4.6). Im Anschluss kann sukzessive über alle Zeilen der Eingabe iteriert werden, um in jedem Schleifendurchlauf zu prüfen, ob eine Leerzeile vorliegt. Gegebenenfalls wird diese an die entsprechende Position in der Ausgabe kopiert. Gleichzeitig werden Block- und Zeilenkommentare im ursprünglichen Flow-Programm gesucht und an die gleiche Stelle in den TypeScript-Code eingefügt. Auf diese Weise werden nach und nach alle Leerzeilen und Kommentare übertragen, sodass die originalgetreue Formatierung hergestellt wird. Schließlich wird Prettier erneut auf die auf diese Weise modifizierte Ausgabe angewandt, um verbleibende Probleme wie beispielsweise doppelte Leerzeilen zu eliminieren.

[31] Siehe Anhang A.2.

5 Auswertung und Diskussion

Nachfolgend sollen nun die Ergebnisse der Migration der zwei Projekte *Components* und *Helios*, die mit Hilfe des umgesetzten Flow-Transpilers durchgeführt wurde, dargelegt werden. Dabei wird zum einen die Erfüllung der Zielvorgaben aus Abschnitt 3.2, zum anderen die Einhaltung der technischen Anforderungen an den Transcompiler aus Abschnitt 3.3 beurteilt und kritisch diskutiert. Damit die erzielten Ergebnisse besser eingeordnet werden können, werden diese hierbei auch, wo möglich, mit den Ansätzen von *Kikura* [87] und *Barabash* [14] zur Transpilierung von Flow verglichen[32]. Zunächst wird jedoch die Durchführung der Migration an sich beschrieben.

5.1 Durchführung der Migration

Bereits während der Entwicklungsphase des Flow-Transpilers wurde dieser immer wieder mit dem Quelltext der vorliegenden Projekte von TeamShirts erprobt, um dessen Praxistauglichkeit anhand einer realen Codebasis zu überprüfen. Nachdem der vollständige Funktionsumfang des Transcompilers hergestellt war, wurde als erstes das Projekt Components, danach das Projekt Helios mit Hilfe des Übersetzers in TypeScript umgewandelt. Diese Reihenfolge wurde bewusst gewählt, da Components keine Abhängigkeit zu anderen Projekten von TeamShirts besitzt, aber Helios wiederum Teile von Components verwendet. Weil Components somit bei der Migration von Helios bereits in TypeScript vorlag, konnte die Typisierung dieser externen Module direkt in die Typüberprüfung von Helios miteinbezogen werden.

Unmittelbar nach Ausführung des Transpilers wurden bei Components 543 und bei Helios 404 neue Typfehler durch den TypeScript-Compiler festgestellt. Diese mussten daraufhin manuell behoben werden, um die Typkorrektheit wiederherzustellen. Die Berichtigung der Typisierung vereinnahmte bei Components sieben und bei Helios acht Personentage. Aufgrund der prinzipiellen Unterschiede der Typsysteme von Flow und Typescript, die im Grundlagenteil bereits dargelegt wurden, war das Auftreten neuer Fehler zu erwarten. Wie beschrieben ist

[32] Vgl. Abschnitt 3.1.2.

beispielsweise die Typinferenz und die Berechnung der Typkompatibilität (nominal versus strukturell) in TypeScript anders umgesetzt als bei Flow.

5.2 Bewertung der Ergebnisse hinsichtlich der Zielvorgabe

Nachdem der Wechsel zu TypeScript vollzogen war, konnten verschiedene Auswertungen durchgeführt werden, um die Ergebnisse bezüglich der Zielsetzung zu bewerten. Im Folgenden sollen die in Kapitel 3 dargelegten Vorgaben hinsichtlich ihrer Erfüllung untersucht werden.

5.2.1 Erkennung weiterer Typ- und Programmfehler

Neu aufgetretene Typfehler

Das primäre Ziel der Migration zu TypeScript war es, weitere Typ- und Programmfehler im bestehenden und zukünftigen Quelltext der Projekte von TeamShirts aufzudecken. Nach der Übersetzung der Flow-Typisierung nach TypeScript wurden sowohl in Components als auch in Helios zahlreiche neue Typfehler festgestellt. Wie bereits ausgeführt bietet der TypeScript-Compiler die Option „`strict`“, um die Strenge der Typüberprüfungen deutlich zu verschärfen, indem beispielsweise die implizite Zuweisung des dynamischen Typs `any` verboten wird [26]. Tabelle 5.1 auf Seite 63 gibt einen Überblick über die Anzahl und die Art der zwölf häufigsten strikten und nicht-strikten Typfehler, die unmittelbar nach der Transpilierung der zwei Projekte erkannt wurden. Insgesamt wurden bei Components 608 bzw. 543 und bei Helios 697 bzw. 404 strikte bzw. nicht-strikte Typverletzungen offengelegt. Im Folgenden soll näher beleuchtet werden, welche Arten von neuen Typfehlern überwiegend aufgedeckt wurden, um aufzuzeigen für welche Klasse von Fehlern Flow und TypeScript zu unterschiedlichen Ergebnissen kommen.

Bei beiden Projekten treten am häufigsten Typfehler auf, die sich darauf beziehen, dass Typen einander nicht zuweisbar sind oder dass Typen ein referenziertes Attribut nicht beinhalten (TS2322, TS2345 bzw. TS2339). Die Gründe für diese Fehler sind vielfältig: Wie dargelegt unterscheiden sich Flow und TypeScript in einigen Aspekten wie der Typinferenz und der Betrachtung von Subtypen grundlegend, sodass hierdurch neue Typverletzungen entstehen

Tabelle 5.1: Auflistung der zwölf häufigsten nach der Transpilierung neu aufgetretene strikten und nicht-strikten TypeScript-Typfehler in den Projekten Components und Helios.

Strikt	Art	Nicht-strikt	Art	Strikt	Art	Nicht-strikt	Art
249	TS2322	184	TS2322	205	TS2307	205	TS2307
104	TS2345	91	TS2345	202	TS2339	88	TS2322
70	TS2339	70	TS2339	147	TS2322	37	TS2345
47	TS2605	48	TS2605	72	TS2345	24	TS2339
39	TS2304	39	TS2304	24	TS2305	24	TS2305
22	TS2741	25	TS2538	15	TS2304	15	TS2304
22	TS2554	23	TS2741	9	TS2531	3	TS2739
12	TS2739	22	TS2554	7	TS2532	3	TS2724
9	TS2307	12	TS2739	4	TS2538	2	TS2605
8	TS2740	9	TS2307	3	TS2724	1	TS2740
5	TS2557	5	TS2557	2	TS2739	1	TS2678
5	TS2305	5	TS2305	2	TS2605	1	TS2374
4	TS1099	4	TS1099	1	TS2740	–	–
COMPONENTS				HELIOS			

Arten von Typfehlern [28]

TS1099 – Die Liste der Typargumente darf nicht leer sein.
TS2304 – Name x kann nicht gefunden werden.
TS2305 – Modul x exportiert kein Element y.
TS2307 – Modul x kann nicht gefunden werden.
TS2322 – Typ x ist Typ y nicht zuweisbar.
TS2339 – Attribut x existiert nicht in Typ y.
TS2345 – Argument mit Typ x ist Parameter mit Typ y nicht zuweisbar.
TS2374 – Doppelte Index-Signatur für Zeichenketten.
TS2531 – Objekt ist möglicherweise `null`.
TS2532 – Objekt ist möglicherweise `undefined`.
TS2538 – Typ x darf nicht als Index-Typ verwendet werden.
TS2554 – x Argumente erwartet, aber y gegeben (Anzahl).
TS2557 – Mindestens x Argumente erwartet, aber y oder mehr gegeben (Anzahl).
TS2605 – Der Typ des JSX-Elements x ist keine Konstruktorfunktion eines JSX-Elements.
TS2678 – Typ x ist nicht vergleichbar mit Typ y.
TS2724 – Modul x exportiert kein Element y. Meinten Sie z?
TS2739 – Typ x fehlen die folgenden Attribute von Typ y: z.
TS2740 – Typ x fehlen die folgenden Attribute von Typ y: z und weiterhin p.
TS2741 – Attribut x fehlt in Typ y, wird aber von Typ z erwartet.

können. Darüber hinaus wurden nach Ausführung des Transpilers externe Typdefinitionen für die eingesetzten Bibliotheken hinzugefügt, um deren korrekte Verwendung zu überprüfen. Weil die bestehende Typisierung von Ausdrücken nicht überall mit diesen Typdeklarationen übereinstimmt, bilden sich auch so neue Typfehler. Bei Helios tritt die Typverletzung TS2307, also dass ein Modul nicht gefunden werden kann, am öftesten auf. Dieser Fehler resultiert daraus, dass in diesem Projekt neben regulären TypeScript-Modulen auch andere Dateitypen wie beispielsweise SVG-Dateien importiert werden, was bei TypeScript normalerweise nicht möglich ist. Um eine derartige Verarbeitung weiterer Dateitypen zu realisieren, wird spezielle Software wie *Webpack* [88] eingesetzt, die eine solche Modulstruktur zu statischen Ausgabedateien transformiert. Durch Anlegen einer globalen Typdeklaration für „Module“ derartiger Dateitypen kann diese Klasse von Typfehlern aber leicht behoben werden (vgl. Quelltext 5.1). Ein weiterer häufiger Fehler ist, dass der Import von Typen aus Bibliotheken fehlschlägt, wenn sich zum Beispiel deren Name in TypeScript geändert hat (TS2305). Außerdem entstehen Typfehler, wenn in Flow globale Typen der Standardbibliothek eingesetzt werden, da diese in TypeScript vor deren Verwendung erst explizit importiert werden müssen (TS2304). Idealerweise sollte der Import dieser zuvor globalen Typen durch den Transpiler, ähnlich zu der in Abschnitt 4.3.3 beschriebenen Übersetzung der React-Typimporte, automatisch durchgeführt werden, um den Aufwand manueller Korrekturen nach der Transpilierung zu reduzieren. Eine weitere Art von Fehler, die vor allem bei Components vermehrt auftritt, betrifft die inkorrekte Typisierung von React-Komponenten (TS2605).

```
1 declare module '*.svg' {
2   export default SVGElement;
3 }
```

Quelltext 5.1: Behebung des Fehlers TS2307 am Beispiel der globalen Deklarationen von SVG-Modulen.

Neu erkannte Programmfehler

Ein Aspekt der Zielsetzung war auch fehlerhaftes Programmverhalten aufzudecken. Während der manuellen Behebung der neu aufgetreten Typverletzungen wurde deshalb untersucht, ob manche der Typfehler tatsächlich semantische Probleme innerhalb des Quelltexts repräsentieren. In der Tat konnten einige Programmfehler durch TypeScript ermittelt werden. Anhand von drei Beispielen soll im Folgenden dargelegt werden, welche Art von Programmfehlern dabei festgestellt wurden. Obwohl keiner der Fälle kritische Laufzeitfehler verursacht, ist es dennoch erstrebenswert, diese zu korrigieren, um die Codequalität insgesamt zu steigern.

Zusätzliche Objektattribute bei Funktionsargumenten Ein Beispiel für derartige Programmfehler ist der inkorrekte Aufruf von Funktionen, die ein Objekt mit verschiedenen Attributen als Argument erwarten. Weil Flow es standardmäßig erlaubt, dass bei Objektliteralen Attribute angegeben werden, die nicht durch den zugrunde liegende Objekttyp spezifiziert sind [64], ist der Fehler bislang unerkannt geblieben. TypeScript behandelt Objekttypen dagegen immer exakt, sodass zusätzliche Attribute eine Typverletzung darstellen und der fehlerhafte Funktionsaufruf damit aufgedeckt wird.

Fehlerhafte Verwendung von React-Komponenten React-Komponenten werden ähnlich wie HTML-Elemente eingesetzt und sind in der Regel durch JSX-Syntax aufgebaut. Mittels selbstdefinierter Attribute können dabei verschiedene Eigenschaften der Komponente festgelegt werden. In einigen Fällen wurde von TypeScript erkannt, dass bei Komponenten Attribute angegeben wurden, die nicht länger bestehen. Weil React-Komponenten intern durch Objekte implementiert werden, die deren Struktur und Attribute beschreiben [2], kann die gleiche Argumentation wie zuvor als Grund für die unterschiedliche Ergebnisse bei Flow und TypeScript herangezogenen werden: Da TypeScript im Gegensatz zu Flow alle Objekte stets exakt betrachtet, stellen zusätzliche angegebene Attribute bei Komponenten eine Typverletzung dar.

Falscher Argumenttyp bei Funktionsaufrufen Zuletzt wurde festgestellt, dass der Typ mancher Funktionsargumente nicht mit dem erwarteten Typ übereinstimmt. Weil die Argumente im aufgetretenen Fall unmittelbar vor Weitergabe an die Funktion durch eine weitere Bibliotheksfunktion umgeformt werden und für diese zweite Funktion in Flow keine Typisierung vorliegt, wurde der Rückgabetyp hier implizit zu `any` umgeformt. Weil `any` jedem Typ zuweisbar ist, wurden die Rückgabewerte der Bibliotheksfunktion als Argument der ersten Funktion akzeptiert, obwohl die Typen in Wahrheit inkompatibel sind. Der Aufruf schlägt zur Laufzeit nur deshalb nicht fehl, weil der Typ des Rückgabewert mittels der dynamischen Typumwandlung von JavaScript implizit angepasst wird. In TypeScript wird dieser Typfehler statisch erkannt, da für die beteiligte Bibliothek eine externe Typdefinition hinzugefügt wurde, sodass die Inkompatibilität der Typen aufgedeckt wird.

Fazit

Es gilt hervorzuheben, dass die Typfehler nach der Migration größtenteils keine Programmfehler darstellen, sondern entweder auf die grundlegenden Unterschiede von Flow und TypeScript

oder auf Unzulänglichkeiten der vorherigen Typisierung durch Flow zurückzuführen sind. Auch die Integration externer Typdefinitionen für die eingesetzten Bibliotheken steigerte zwar die Abdeckung der Projekte durch das Typsystem, aber gleichzeitig wurden hierdurch zum Teil neue Typfehler eingeführt. Dass mit dem Wechsel zu TypeScript eine hohe Zahl zuvor unerkannter kritischer Laufzeitfehler in den Projekten gefunden wird, war aufgrund der kontinuierlichen maschinellen und menschlichen Softwaretests unwahrscheinlich, da angenommen werden kann, dass fatale Fehler im Allgemeinen ohnehin bereits im Vorfeld entdeckt worden wären. Wie dargelegt konnten durch den Wechsel zu TypeScript aber dennoch einerseits einige tatsächliche unkritische Programmfehler aufgedeckt, andererseits allgemeine Schwächen der bisherigen Typisierung offengelegt werden. Deshalb kann die Zielsetzung insgesamt als erfüllt betrachtet werden.

5.2.2 Unterstützung externer Bibliotheken

Die Migration zu TypeScript wurde weiterhin angestrebt, weil vermutet wird, dass hier Typdeklarationen für eine insgesamt größere Zahl von JavaScript-Bibliotheken vorliegen und diese aktueller sind als bei Flow. Im Folgenden soll beleuchtet werden, ob dies tatsächlich zutrifft. Wie ausgeführt stellen einige Projekte selbst Definitionsdateien mit der Typisierung bereit, bei anderen kann auf separat gepflegte Typisierungen zurückgegriffen werden.

Für Flow und TypeScript existiert jeweils ein Projekt mit dem Ziel, Typdeklarationen für die gängigsten JavaScript-Bibliotheken bereitzustellen. Während seit 2012 für TypeScript das Projekt *DefinitelyTyped* [34] besteht, ist 2017 für Flow ein vergleichbarer Ansatz namens *flow-typed* [66] entstanden. Im Allgemeinen kann angenommen werden, dass Bibliotheken, die ihre eigenen Typdeklarationen mitliefern, aktueller und korrekter sind, weil im Zuge der Veröffentlichung neuer Versionen in der Regel auch die Typisierung durch den Autor entsprechend aktualisiert wird[33]. Bei der nachträglichen Erstellung von Typdeklarationen durch die Entwicklergemeinschaft besteht die Gefahr, dass die Typisierung das Verhalten der Bibliothek nicht richtig abbildet oder dass diese veralten. Deshalb sind Typdefinitionen, die von den Projekten selbst verwaltet werden, generell zu bevorzugen. Zunächst sollen DefinitelyTyped und flow-typed in Tabelle 5.2 mittels verschiedener Kennzahlen allgemein verglichen werden, um die Verfügbarkeit von gemeinschaftlich gepflegten Typdeklarationen

[33] Im Falle von TypeScript können die Definitionsdateien beispielsweise auch durch den TypeScript-Compiler automatisch erzeugt werden [26].

zu untersuchen. Wie die Zahl der Git-Commits und der Beitragenden im Jahresmittel belegen, weist DefinitelyTyped eine deutlich größere Aktivität als flow-typed auf. Insgesamt werden bei TypeScript in etwa zehn mal mehr Bibliotheken unterstützt als bei Flow, was trotz des Umstands, dass DefinitelyTyped fünf Jahre älter ist, einen beachtlichen Faktor darstellt.

Tabelle 5.2: Verschiedene Eigenschaften der Projekte *DefinitelyTyped* und *flow-typed* auf der Software-Plattform GitHub [122] (Stand: Oktober 2019).

Projekt	**Typdefinitionen**	**Commits**		**Beitragende**		**Sterne**
flow-typed	628	2.600	520 p. a.	800	201 p. a.	3.400
DefinitelyTyped	6202	65.500	8188 p. a.	9.800	1785 p. a.	25.000

Quelle der Aktivitätsdaten: Analyse der Git-Repositories mit *GitStats* [77].
Angaben pro Jahr (p. a.) stellen den Durchschnitt aller Jahre dar.

Nachfolgend wird nun die Unterstützung externer Bibliotheken anhand der 15 am häufigsten eingesetzten Pakete innerhalb der zwei Projekte von TeamShirts in Tabelle 5.3 konkret betrachtet. Die Bibliotheken, die am öftesten verwendet werden, wurden durch statische Analyse des Quelltexts ermittelt, indem gezählt wurde, wie oft diese jeweils importiert werden. Für diese Auswahl wurde anschließend untersucht, ob einerseits für Flow und TypeScript generell Typdeklarationen bestehen, andererseits ob diese als Bestandteil der Bibliothek oder separat erstellt vorliegen. Weiterhin wurde beleuchtet, ob die Typdefinitionen der aktuellen Version des Pakets entsprechen oder veraltet sind. Dabei wird nur die Haupt- und Nebenversionsnummer[34] der Pakete betrachtet, weil die Revisionsnummer nur der Behebung von Programmfehlern dient und per Definition die Schnittstellen nicht verändern darf. Bei flow-typed wird die Kompatibilität der Typdefinitionen allerdings nur für die Hauptversion angegeben, sodass unklar ist, inwieweit die Deklarationen tatsächlich alle Funktionen der aktuellen Nebenversion abbilden. DefinitelyTyped differenziert hier stärker und gibt jeweils für alle Pakete exakte Versionsnummern an, wodurch die Aktualität besser eingeschätzt werden kann.

Während bei Flow für vier der 15 Bibliotheken keine Typdefinition vorliegt, existiert bei TypeScript lediglich in einem Fall keine Typisierung. Der Großteil der Deklarationen entstammt jeweils den Projekten flow-typed bzw. DefinitelyTyped. Nur in einem bzw. drei Fällen bei Flow bzw. TypeScript sind die Typdefinitionen direkt in die Pakete integriert. Für den Großteil der Bibliotheken sind die Typisierungen sowohl bei Flow als auch bei TypeScript aktuell.

[34] Vgl. *Semantic Versioning* [112].

Tabelle 5.3: Verfügbarkeit von Typdeklarationen für Flow bzw. TypeScript für die 15 am häufigsten verwendeten JavaScript-Bibliotheken innerhalb der Projekte Components und Helios.

Bibliothek	**Version**	**Verwendung**	**Verfügb. Flow**		**Verfügb. TypeScript**	
`react`	16.11	388	16.11	I	16.11	E
`styled-components`	4.4	135	4.x	E	4.1	E
`@storybook/react`	5.2	74	5.x	E	4.0	E
`chai`	4.2	73	4.x	E	4.2	E
`react-redux`	7.1	72	7.x	E	7.1	E
`react-apollo`	3.1	34	–		3.1	I
`react-router-dom`	5.1	22	5.x	E	5.1	E
`react-loadable`	5.5	17	5.x	E	5.5	E
`lodash`	4.17	15	4.x	E	4.14	E
`react-router`	5.1	13	5.x	E	5.1	E
`es6-error`	4.1	9	4.x	E	4.1	I
`react-motion`	0.5	5	–		0.29	E
`axios`	0.19	3	0.19	E	0.19	I
`react-collapse`	5.0	3	–		4.0	E
`css-math`	0.4	2	–		–	

`E`: Separat gepflegte Typdefinition (durch *flow-typed* bzw. *DefinitelyTyped*)
`I`: In Bibliothek integrierte Typdefinition

Nur bei TypeScript besteht in zwei bzw. drei Fällen eine Abweichung von der Haupt- bzw. Nebenversion. Aus den dargelegten Gründen kann nicht überprüft werden, ob auch bei Flow veraltete Typdefinitionen hinsichtlich der Nebenversion vorliegen, weil diese durch flow-typed nicht angegeben wird. Da bei TypeScript für die gegebenen Projekte von TeamShirts eine größere Zahl von Bibliotheken durch Typdeklarationen abgebildet werden und diese in höherem Maße in die Pakete integriert sind, kann die Zielsetzung, externe Bibliotheken besser zu unterstützen, insgesamt als erreicht betrachtet werden. Kritisch anzumerken ist aber, dass bei TypeScript für fünf Bibliotheken keine vollständig aktuelle Version vorliegt. Jedoch ist dies auch bei Flow nicht gesichert.

5.2.3 Performance der Typüberprüfungen

Messung der Laufzeit der Typüberprüfungen

Eine weitere Zielsetzung des Wechsels zu TypeScript war es, die Performance der Typüberprüfungen zu verbessern, das heißt deren Laufzeit zu verringern. Nachdem die Migration der zwei Projekte abgeschlossen war, konnten die Laufzeiten der vollständigen Typüberprüfung durch Flow bzw. TypeScript ermittelt werden. Auch die Messung der Zeitspanne von inkrementellen Überprüfungen durch den Flow- bzw. TypeScript-Sprachserver wurde durch Auswertung der Logdateien in Betracht gezogen. Jedoch ist dieser Ansatz nur möglich, wenn ein Editor ausgeführt wird, der den jeweiligen Server startet und daraufhin Anfragen bezüglich der Typisierung an diesen stellt. Weil die Ergebnisse somit stark von der Implementierung des Editors abhängen, besteht die Gefahr, dass diese verfälscht werden und wenig aussagekräftig sind. Deshalb werden inkrementelle Typüberprüfungen nachfolgend nicht betrachtet.

Zur Bestimmung der Laufzeit der vollständigen Berechnung wurden in beiden Projekten für Flow und TypeScript jeweils 100 Proben (*Samples*) mit Hilfe des GNU-Programms *time* [85] gemessen und die zehn kleinsten und größten Werte daraufhin verworfen, um den Einfluss von „Ausreißern“ zu minimieren. Aus den verbleibenden 80 Werten wurde anschließend der Mittelwert gebildet und die Standardabweichung berechnet. Dabei wurde in allen Messungen die zum damaligen Zeitpunkt aktuelle Version 3.5 von TypeScript und die von TeamShirts eingesetzte Version 0.96 von Flow verwendet. Um auch den Einfluss von unterschiedlich leistungsfähiger Hardware miteinzubeziehen, wurden die Messreihen auf vier verschiedenen Systemen durchgeführt:

A. AMD Phenom II X6 1055T Prozessor mit 2,9 GHz[35] und 6 Rechenkernen (2010)
   16 GB Arbeitsspeicher, Solid State Drive, Arch Linux
B. Intel Core i5-4258U Prozessor mit 2,4 GHz und 4 Rechenkernen (2013)
   8 GB Arbeitsspeicher, Solid State Drive, Arch Linux
C. Intel Core i5-4210M Prozessor mit 2,6 GHz und 4 Rechenkernen (2014)
   16 GB Arbeitsspeicher, Solid State Drive, Arch Linux
D. Intel Core i7-6700 Prozessor mit 3,4 GHz und 8 Rechenkernen (2015)
   32 GB Arbeitsspeicher, Solid State Drive, Debian Linux

[35] Es wird jeweils der Grundtakt des Prozessors und nicht der maximal mögliche Wert durch dynamische Übertaktung angegeben.

Die durch diese Methodik ermittelten Messwerte, deren Standardabweichung und die relative Veränderung der Laufzeiten werden in Tabelle 5.4 für beide Projekte gezeigt: Offensichtlich beschleunigen modernere, performante Prozessoren mit höherer Taktfrequenz und größeren Caches generell die Typüberprüfung durch Flow bzw. TypeScript. Die These, dass TypeScript schneller als Flow sei, lässt sich aber nur anhand des Projekts Helios für die Systeme B, C und D belegen (vgl. negative Werte für relative Veränderung der Laufzeit). Bei Components ist TypeScript dagegen stets deutlich langsamer als Flow.

Tabelle 5.4: Durchschnittliche Laufzeit in Sekunden, Standardabweichung (s) und relative Veränderung der Zeitspanne (rel. Δ) der vollständigen Typüberprüfung der Projekte Components und Helios durch Flow 0.96 und TypeScript 3.5 auf verschiedenen Hardware-Systemen (S).

	Flow		**TypeScript**			**Flow**		**TypeScript**		
S	Laufzeit	s	Laufzeit	s	rel. Δ	Laufzeit	s	Laufzeit	s	rel. Δ
A	7,87	0,09	15,50	0,08	97,0%	12,20	0,16	13,06	0,70	7,1%
B	6,86	0,05	10,59	0,06	54,4%	10,90	0,33	8,49	0,16	-22,1%
C	6,50	0,02	9,56	0,05	47,1%	11,70	0,07	7,63	0,04	-34,8%
D	3,38	0,02	6,41	0,04	89,6%	5,15	0,02	4,94	0,03	-4,1%
	COMPONENTS					HELIOS				

Wie im Grundlagenteil bereits ausgeführt, wird die Berechnung der Typkorrektheit durch Flow stark parallelisiert, um deren Laufzeit zu verringern. TypeScript bietet keine Unterstützung für eine nebenläufige Typüberprüfung durch mehrere Threads [99], sodass die Performance hier vorrangig von der Leistungsfähigkeit der einzelnen Prozessorkerne abzuhängen scheint. Diese These wird durch die gemessenen Daten unterstützt, denn je performanter die Rechenkerne sind, desto kleiner wird die Differenz zwischen den Laufzeiten von Flow und TypeScript bei Components bzw. desto größer wird diese bei Helios. Für System D scheint jedoch die Parallelisierung von Flow aufgrund der hohen Zahl von acht Prozessorkernen den größeren Einfluss zu haben, als die Leistungsfähigkeit der einzelnen Kerne. Weil die Systeme B und C jeweils nur über vier Rechenkerne verfügen, ist der Effekt der Nebenläufigkeit hier kleiner. Als einen möglichen Grund, warum Components im Vergleich zu Helios durch Flow insgesamt schneller überprüft werden kann, kann angeführt werden, dass dieses Projekt aus einer Vielzahl unabhängiger Komponenten besteht deren Verarbeitung somit gut durch Flow parallelisiert werden kann. In Helios besteht eine größere Abhängigkeit der Module untereinander.

Einfluss von Parallelisierung auf die Laufzeiten

Der Einfluss der Parallelisierung auf die Laufzeit der Berechnungen soll im Folgenden näher beleuchtet werden. Hierfür wurde eine weitere Testreihe durchgeführt, in der die Zahl der einsetzbaren Rechenkerne für die Berechnung durch das Linux-Programm *taskset* [90] eingeschränkt wird. Dabei wird Flow bzw. TypeScript durch den Scheduler des Betriebssystems zunächst lediglich ein Rechenkern, dann sukzessiv immer ein weiterer zugeordnet. Abbildung 5.1 auf Seite 72 zeigt jeweils ein Diagramm für Components und Helios, welches die Beeinflussung des Laufzeitverhaltens durch diese Einschränkung grafisch darstellt. Die exakten Messwerte, deren Standardabweichung und die relative Veränderung werden darüber hinaus in Tabelle 5.5 aufgeführt.

Tabelle 5.5: Durchschnittliche Laufzeit in Sekunden, Standardabweichung (s) und relative Veränderung der Zeitspanne (rel. Δ) der vollständigen Typüberprüfung der Projekte Components und Helios durch Flow 0.96 und TypeScript 3.5 bei zunehmender Zahl verfügbarer Rechenkerne (K). Gemessen mit Intel Core i7-6700 CPU mit 3,4 GHz (System D).

	Flow		**TypeScript**			**Flow**		**TypeScript**		
K	Laufzeit	s	Laufzeit	s	rel. Δ	Laufzeit	s	Laufzeit	s	rel. Δ
1	10,59	0,03	10,13	0,03	-4,3%	17,26	0,05	8,81	0,01	-49,0%
2	6,19	0,02	6,85	0,05	10,7%	9,61	0,04	5,44	0,04	-43,4%
3	4,74	0,03	6,50	0,02	37,1%	7,23	0,03	5,03	0,02	-30,4%
4	4,05	0,03	6,43	0,02	58,8%	6,16	0,04	4,95	0,01	-19,6%
5	3,79	0,04	6,48	0,02	71,0%	5,82	0,03	5,04	0,15	-13,4%
6	3,63	0,02	6,41	0,04	76,6%	5,51	0,03	4,94	0,03	-10,3%
7	3,52	0,02	6,40	0,03	81,8%	5,31	0,03	4,93	0,03	-7,2%
8	3,42	0,02	6,40	0,03	87,1%	5,16	0,02	4,94	0,03	-4,3%
	COMPONENTS					HELIOS				

Sowohl bei Flow als auch bei TypeScript wird die Laufzeit der Berechnung bereits bei Verwendung von zwei Rechenkernen im Vergleich zu einem einzigen Kern deutlich verringert. Obwohl TypeScript eine parallelisierte Kompilierung grundsätzlich nicht unterstützt, wird auch dieser Prozess beschleunigt, weil der TypeScript-Compiler durch Node.js ausgeführt wird und diese Umgebung standardmäßig mehrere Threads startet, um beispielsweise teure Ein- und Ausgabeoperationen auf weitere Prozessorkerne auszulagern [33]. Für mehr als drei Kerne bleibt die Laufzeit von TypeScript daraufhin in beiden Projekten aber in etwa konstant.

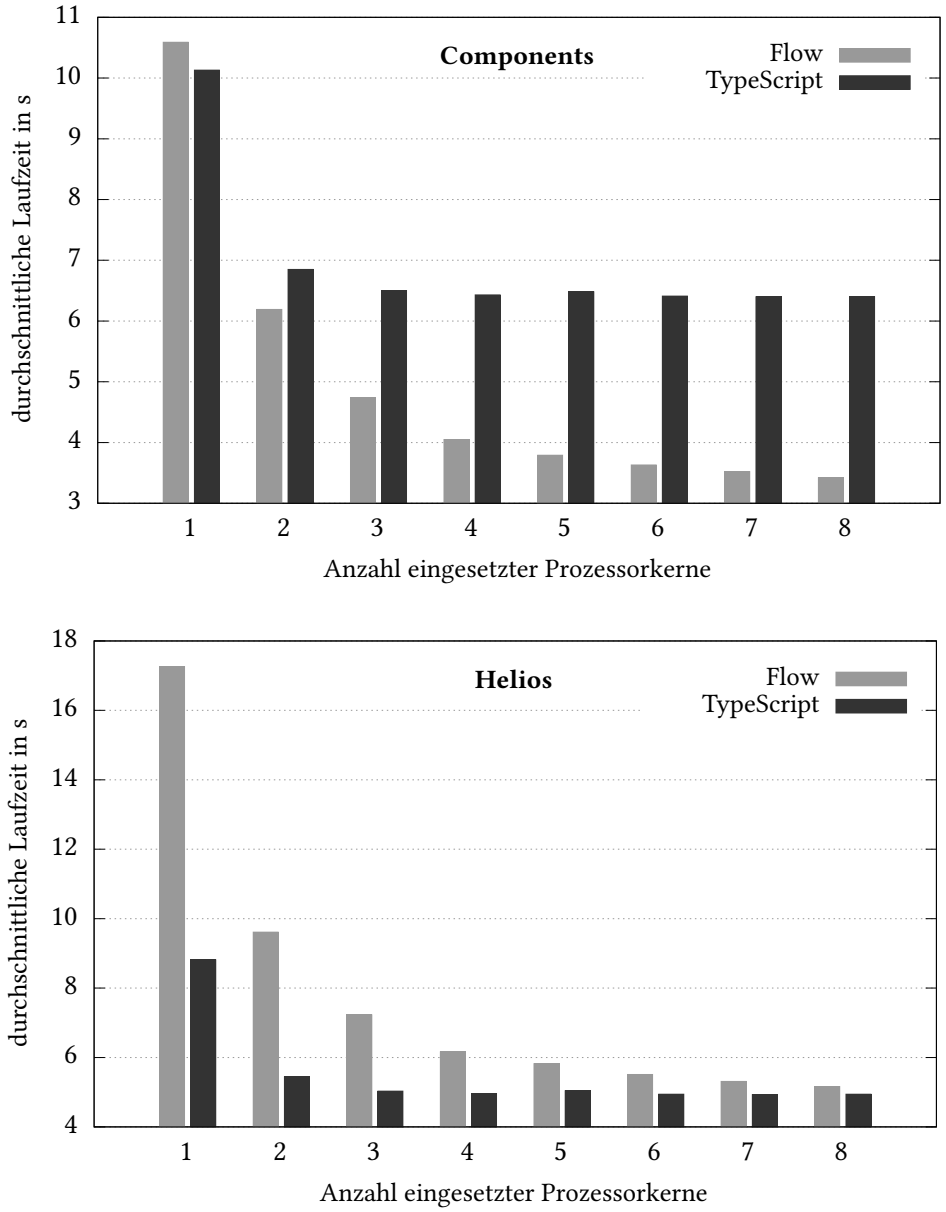


Abbildung 5.1: Einfluss der zur Verfügung stehenden Rechenkerne auf durchschnittliche Laufzeit der Typüberprüfung von Flow 0.96 und TypeScript 3.5 der Projekte Components und Helios.

Gemessen mit Intel Core i7-6700 CPU mit 3,4 GHz (System D).

Wie die Messwerte und Diagramme zeigen, verringert sich die Laufzeit von Flow dagegen mit wachsender Zahl zur Verfügung stehender Rechenkerne aufgrund der Unterstützung von Multithreading stetig. Ab etwa fünf Kernen sind die Zugewinne allerdings nur noch gering, was auf den zunehmend größeren Overhead durch die notwendige Synchronisierung der Threads zurückgeführt werden könnte. Bei maximal möglicher Parallelisierung durch acht Kerne erreicht Flow bei Helios nahezu die Laufzeit von TypeScript.

Fazit

Die angestrebte Verbesserung der Performance durch die Migration zu TypeScript kann somit im vorliegenden Fall nur als teilweise erreicht betrachtet werden, da nur in drei Fällen bei Helios tatsächlich geringere Laufzeiten für eine vollständige Typüberprüfung gemessen wurden. Die stark parallelisierte Architektur von Flow scheint der von TypeScript bezüglich der Geschwindigkeit in den meisten Fällen überlegen. Es wird aber vermutet, dass die Performance von TypeScript gesteigert werden könnte, wenn auch hier die Berechnung der Typkorrektheit nebenläufig wäre. Laut Aussage eines Entwicklers von TypeScript im März 2019 wird die Umsetzung von Multithreading im TypeScript-Compiler bereits in Betracht gezogen [117].

5.2.4 Zukunftssicherheit und Transparenz der Technologie

Das letzte Ziel der Migration zu TypeScript war schließlich die Zukunftssicherheit zu steigern, da angenommen wird, dass die Entwicklung von TypeScript transparenter abläuft als bei Flow. Hierzu wurden in Abschnitt 3.2.4 verschiedene Fragestellungen aufgeworfen, die im Folgenden beantwortet werden sollen. Der erste Aspekt betrifft die Verfügbarkeit eines öffentlich einsehbaren Projektplans (*Roadmap*) in welchem strategische Entscheidungen kommuniziert werden, sodass die langfristige Weiterentwicklung der Technologie besser eingeschätzt werden kann. Für Flow existiert derzeit keine solche Roadmap: Eine entsprechende Anfrage im November 2017 auf GitHub blieb durch das Entwicklerteam von Flow unbeantwortet [108]. Ein Mitarbeiter von Facebook bestätigte aber im Januar 2019, dass beabsichtigt werde zukünftig einen Projektplan für Flow zu veröffentlichen [65]. Die Entwicklung von TypeScript ist an dieser Stelle deutlich transparenter: So werden auf GitHub einerseits perspektivische Ziele der Programmiersprache kommuniziert, andererseits konkrete Halbjahrespläne ausführlich

dargelegt [107]. Benutzer von TypeScript können damit sehr gut beurteilen, welche Funktionen in Zukunft zu erwarten sind und wie diese priorisiert werden.

Sowohl Flow als auch TypeScript werden quelloffen auf der Plattform GitHub entwickelt, sodass Benutzer die Möglichkeit haben, Probleme (*Issues*) zu melden und sich in die Weiterentwicklung der Systeme einzubringen. Um zu untersuchen, wie schnell derartige Tickets bearbeitet werden, wurden für Flow und TypeScript entsprechende Daten durch die Programmierschnittstelle[36] von GitHub [27] erhoben. Anhand von 3.800 bzw. 20.000 Einträgen von Flow und TypeScript wurde ermittelt, wie viel Zeit im Median zwischen der Erstellung und der Behebung eines Problemberichts vergeht: Hierbei ergibt sich für Flow ein Wert von 11,58 und für TypeScript ein Wert von 6,61 Tagen. Zwar muss das Schließen eines Tickets nicht in jedem Fall bedeuten, dass tatsächlich ein Programmfehler vorlag und dieser korrigiert wurde, dennoch kann bei TypeScript somit insgesamt eine schnellere Bearbeitung von gemeldeten Problemen festgestellt werden.

Die letzte Fragestellung aus Abschnitt 3.2.4 betrifft die Zukunftssicherheit von Flow bzw. TypeScript, also ob die langfristige Unterstützung der Systeme durch die Autoren Facebook bzw. Microsoft gewährleistet ist. Weil keine Roadmap für Flow vorliegt und bekannt wurde, dass selbst interne Projekte bei Facebook wie zum Beispiel *Jest* [53] zu TypeScript migrieren [1], bestand bei einigen Entwicklern Verunsicherung darüber, ob Facebook die Fortentwicklung des Systems weiterhin fokussiert. Diese Zweifel wurden jedoch von einem der Autoren von Flow, Avik Chaudhuri, im bereits erwähnten Artikel „*What the Flow team has been up to*“ [23] ausgeräumt. Er betont, dass Facebook intern nach wie vor primär Flow einsetze und die Zukunft des Systems gesichert sei. Im Fall von TypeScript besteht wie ausgeführt eine öffentlich einsehbarer Projektplan, sodass von einer fortwährenden Unterstützung durch Microsoft ausgegangen werden kann.

Zusammenfassend kann gesagt werden, dass die Migration zu TypeScript tatsächlich Vorteile bezüglich der Zukunftssicherheit und Transparenz bietet. Im Vergleich zu Flow besteht hier eine deutlich höhere Planungssicherheit bezüglich der Weiterentwicklung des Typsystems, weil eine Roadmap öffentlich zugänglich ist. Auch Fehlerberichte auf GitHub werden bei TypeScript insgesamt schneller bearbeitet. Die langfristige Unterstützung der Systeme durch die ursprünglichen Autoren ist in beiden Fällen anzunehmen.

[36] Application Programming Interface (API).

5.3 Erfüllung der technischen Anforderungen

Nachdem die Erfüllung der Zielvorgaben durch den Wechsel zu TypeScript geprüft wurden, soll im Folgenden auch die Einhaltung der technischen Anforderungen an den Flow-Transpiler untersucht werden.

5.3.1 Äquivalente und vollständige Übersetzung der Flow-Typen

Äquivalenz der Übersetzungen

Als erste und wichtigste technische Anforderung an den Transpiler wurde die äquivalente und vollständige Übersetzung der gesamten Flow-Syntax nach TypeScript definiert. Die bedeutungsgleichen TypeScript-Ausdrücke der verschiedenen Flow-Typen konnten dabei mehrheitlich einfach gefunden werden, weil TypeScript einen sehr ähnlichen Funktionsumfang wie Flow besitzt und sich oftmals nur die Schlüsselwörter unterscheiden[37]. Kompliziertere, nicht offensichtliche Transformationen wurden experimentell bestimmt.

```
Transpiling example.js...

  1 | // @flow
> 2 | const existentialType: * = String(2 * 3);
    |                        ^
  Warning: Flow's Existential Type (*) is not expressible in TypeScript.
           It will be replaced with 'any'.
           See https://github.com/Microsoft/TypeScript/issues/14466.
```

Quelltext 5.2: Ausgabe von Warnungen bei Autreten nicht äquivalent übersetzbarer Flow-Typen.

Für einige, wenige Typen existiert keine absolut bedeutungsgleiche Übersetzung, da TypeScript manche der Funktionen von Flow nicht unterstützt. Infolgedessen kommt es hier bei der Transpilierung zwingend zu einem Verlust von Typinformation, was in neuen Typfehlern resultieren kann. Die inkorrekte Transformation derartiger Typen wird gemäß Anforderung 3.3.1 akzeptiert, jedoch muss der Benutzer bei Auftreten eines solchen Falls gewarnt werden. Quelltext 5.2 zeigt wie eine solche Warnung aufgebaut ist. Dabei wird einerseits die genaue Position des verursachenden Flow-Typs im Quelltext angegeben, andererseits wird der Benutzer über die

[37] Vgl. Übersetzungstabellen in Abschnitt 4.3.2.

Hintergründe der Warnung durch einen Link informiert. Nachfolgend wird dargelegt, welche der Funktionen von Flow nicht absolut korrekt übersetzt werden können und wie dies jeweils innerhalb des Transpilers gehandhabt wird.

- **Existential type**
  In TypeScript existiert kein Typ, der Flows *Existential type* entspricht [46]. Wie Quelltext 5.2 bereits andeutet, wird dieser Typs deswegen durch den Typ *any* ersetzt. Weil *any* Supertyp jeden Typs ist, geht damit Typinformation verloren, sofern Flow zuvor einen konkreteren Typ inferieren konnte. Jedoch gibt die Dokumentation von Flow an, dass dies oftmals nicht funktioniert und der *Existential type* deshalb in diesen Fällen ohnehin äquivalent zu *any* ist [54].
- **Index-Signaturen in Objekttypen**
  Durch eckige Klammern kann in Objekttypen ein Index, also die Abbildung eines Typs von Namen auf Werte eines anderen Typs, angegeben werden[38]. Flow unterstützt hierbei jeden beliebigen Typ für die Attributnamen, TypeScript hingegen nur `string` und `number` [80]. Infolgedessen entfernt der Transpiler Index-Signaturen, die von diesen zugelassenen Typen abweichen, und fügt stattdessen jeweils eine Signatur für `string` und `number` ein, um die größtmögliche Menge von Schlüsseltypen zu erlauben.
- **Opaque type**
  Auch opake Typen werden durch TypeScript nicht unterstützt [118]. Deshalb wird dieser Typ bei der Übersetzung durch ein Typalias ersetzt, das auf den gleichen Typ verweist.
- **Rückgabewert von Konstruktoren**
  Flow ermöglicht es, den Rückgabewert von Konstruktorfunktionen explizit anzugeben. Dies ist in TypeScript bewusst verboten, weil es im Allgemeinen als schlechter Programmierstil gilt, wenn ein Konstruktor etwas anderes als eine Klasseninstanz zurückliefert [124]. Ein solcher Rückgabewert wird deswegen durch den Transpiler entfernt, damit keine fehlerhafte TypeScript-Syntax entsteht.
- **Varianz**
  Ein letzter Aspekt von Flow, der in TypeScript in einigen Fällen nicht abbildbar ist, ist die Varianz von Typen. Weil die Syntax von TypeScript nur Kovarianz (Schlüsselwort `readonly`) unterstützt, werden die Varianz-Signaturen von Flow für alle anderen Fälle verworfen. Dies beinhaltet neben Objektattributen beispielsweise auch Typparameter, die in Flow entsprechend annotiert werden können.

[38] Vgl. Tabelle 2.1.

Neben diesen Flow-Funktionen, die prinzipiell nicht bedeutungsgleich übersetzt werden können, konnte darüber hinaus für die drei Hilfstypen *Object map*, *Object map with key* und *Tuple map*, wie ausgeführt, kein TypeScript-Ausdruck gefunden werden, der diesen entspricht. Deshalb ist die äquivalente Transformation der Typisierung auch in diesen drei Fällen nicht gegeben, weil diese Typen durch *any* ersetzt werden. Jedoch kann angenommen werden, dass diese Hilfstypen in der Regel nur einen kleinen Teil der Typisierung ausmachen, sodass der Verlust von Typinformation insgesamt gering ist. Abgesehen von den beschriebenen Ausnahmen wurde die Äquivalenz der Übersetzungen damit erreicht.

Vollständigkeit der Transformationen

Wie bereits in Abschnitt 3.3.1 dargelegt ist der vollständige Funktionsumfang der Implementierung durch die Spezifikation des Parsers [92] von Babel präzise definiert, weil das Dokument genau festlegt, welche Knoten des abstrakten Syntaxbaums Flow-Syntax darstellen. Sofern sämtliche dieser Elemente korrekt in ihr TypeScript-Gegenstück transformiert werden, ist die Vollständigkeit erzielt. Dies setzt vorraus, dass der abstrakte Syntaxbaum von Babel tatsächlich jegliche Flow-Syntax abbildet. Es konnte kein Beleg dafür gefunden werden, dass diese Annahme falsch ist.

Ein weiterer Aspekt, der zur Erzielung einer vollständigen Umsetzung beiträgt, ist die Typisierung von Babel, da diese zum Beispiel den Vereinigungstyp `Flow` bereitstellt, der alle Knotentypen von Flow umfasst [13]. Somit kann durch eine geeignete Typisierung statisch überprüft werden, ob das Programm alle Elemente eines solchen Vereinigungstyps verarbeitet. Auch hier muss die Vorraussetzung gelten, dass die Typisierung von Babel bezüglich Flow korrekt ist. Der umgesetzte Transpiler verwendet diesen Ansatz beispielsweise innerhalb der zentralen Umwandlungsfunktion für beliebige Flow-Typen (`convertFlowType`[39]), sodass statisch garantiert werden kann, dass dabei ausnahmslos alle Elemente des Vereinigungstyps `FlowType` transformiert werden.

Die Erzielung der Korrektheit und Vollständigkeit der Transformationen wird durch den in Abschnitt 4.2.1 dargelegten Ansatz der testgetriebenen Entwicklung gewährleistet. In der Dokumentation von Flow [58] wird die Syntax aller Sprachkonstrukte beschrieben, sodass bekannt ist, welche Typen prinzipiell möglich sind. Durch Anlegen von Fixture-Dateien kann

[39] Vgl. Quelltext 4.5.

damit die korrekte Übersetzung sämtlicher Flow-Funktionen umfangreich getestet werden. Wie ausgeführt sind insgesamt 1022 Testfälle entstanden, um alle Basis- und Hilfstypen sowie Typdeklarationen zu überprüfen. Durch eine hohe Testabdeckung von 93% wird darüber hinaus sichergestellt, dass die verschiedenen Programmverzweigungen des Transpilers tatsächlich die angestrebte Funktionalität umsetzen. Die Vollständigkeit der Programmtransformation wurde somit erreicht.

Vergleich mit Ansätzen von Kikura und Barabash

Die zwei betrachteten Aspekte der Äquivalenz und Vollständigkeit der Transformationen sollen auch für die Ansätze von Kikura und Barabash beleuchtet werden, um das erzielte Ergebnis einzuordnen. Dabei wird untersucht, ob diese Plugins bei Anwendung auf die angelegten Fixture-Tests zur Erprobung der korrekten Übersetzung der Flow-Typisierung zu den gleichen Ergebnissen kommen. Tabelle 5.6 auf Seite 79 zeigt, bei welchen Testfälle hierbei eindeutig eine fehlerhafte Transformation vorgenommen wird. Darunter wird einerseits die semantisch inkorrekte Übersetzung eines Typs, andererseits unzulässige TypeScript-Syntax in der Ausgabe verstanden. Eine solche invalide Syntax entsteht beispielsweise dann, wenn Spezialfälle nicht richtig behandelt werden. Weil sich die Tabelle am Aufbau der Fixture-Tests orientiert, werden nicht alle einzelnen Typen wie in Kapitel 2 separat, sondern gruppiert, aufgelistet. Die Kategorie „*Primitives*“ enthält dabei zum Beispiel die Tests für alle primitiven Flow-Typen wie *Number type*, *String literal type* et cetera.

Die Übersetzung der Flow-Typen durch Kikura ist in 98,1% der Fälle korrekt. Jedoch können dabei nur 17 der 31 Testfälle überhaupt betrachtet werden, weil die Transpilierung der verbleibenden 14 Eingabedateien gänzlich fehlschlägt, da hier jeweils Laufzeitfehler auftreten (Symbol ↯). Somit ist nicht nachvollziehbar, ob eventuell weitere Fehler vorliegen. Auch bei Barabash stürzt das Programm bei drei der Fixture-Dateien ab, sodass diese Tests nicht bewertet werden können. In 89,0% der anderen Fälle wird die Transformation hier aber korrekt umgesetzt. Die Gründe, warum die Übersetzungen bei Kikura und Barabash in einigen Fällen falsch sind, sind vielfältig. Im Folgenden werden einige der Ursachen aufgelistet:

- Wie ausgeführt dürfen Konstruktoren in TypeScript keine Rückgabewerte deklarieren. Sofern diese, wie hier bei Barabash, nicht entfernt werden, entsteht ein Syntaxfehler.

Tabelle 5.6: Fehlerhafte Übersetzung von Flow-Typen bei Kikura [87] und Barabash [14] im Vergleich zur vorliegenden Implementierung (↯ = Programmabsturz).

		KIKURA		BARABASH		VORL. IMPL.
Typ	**Testfälle**	**Fehler**	**rel. Anteil**	**Fehler**	**rel. Anteil**	**Fehler**
Any type	8	0	0	0	0	0
Array type	13	0	0	2	15,5%	0
Class type	43	↯	–	4	9,3%	0
Empty type	7	↯	–	0	0	0
Function type	34	↯	–	4	11,8%	0
Generics	17	↯	–	1	5,9%	0
Interface type	61	↯	–	↯	–	0
Intersection type	29	↯	–	2	6,9%	0
Maybe type	27	↯	–	23	85,2%	0
Mixed type	6	0	0	0	0	0
Module type	32	↯	–	6	18,8%	0
Object type	122	↯	–	↯	–	0
Opaque type	10	2	20,0%	6	60,0%	0
Primitives	9	0	0	0	0	0
This type	8	0	0	↯	–	0
Tuple type	5	0	0	1	20,0%	0
Type alias	22	↯	–	3	13,6%	0
Type cast	16	0	0	0	0	0
Type declarations	44	↯	–	14	31,8%	0
Typeof type	20	3	15,0%	2	10,0%	0
Union type	19	↯	–	1	5,3%	0
Call	6	↯	–	0	0	0
Difference	11	0	0	0	0	0
Element type	13	0	0	0	0	0
Exact	7	0	0	1	14,3%	0
Existential type	5	0	0	0	0	0
Keys	5	0	0	0	0	0
None maybe type	6	4	66,7%	0	0	0
Property type	8	0	0	0	0	0
Rest	13	↯	–	0	0	0
Shape	11	3	22,3%	0	0	0
Insgesamt	**637**	**12**	**1,9%**	**70**	**11,0%**	**0**

- Flow erlaubt es, Funktionsparameter, die mit einem Standardwert [36] initialisiert werden, als optional zu markieren[40]. Derartige Parameter dürfen in TypeScript nicht optional sein, sodass die Notation durch den Transpiler entsprechend angepasst werden muss.
- Bei Barabash wird der *Nullable type* (*Maybe type*) falsch übersetzt. Statt der Vereinigung aus dem angegeben Typ, `null` und `undefined` wird hier der Typ lediglich mit `null` kombiniert.
- Kikura wandelt opake Typen wie die vorliegende Umsetzung in reguläre Typaliase um. Jedoch werden dabei bestehende Typparameter verworfen.
- Schließlich werden bei beiden Ansätzen einige der Hilfstypen überhaupt nicht transformiert und einfach in die Ausgabe übernommen. Weil diese Typen in TypeScript undefiniert sind, entsteht auch so ein Fehler.

Insgesamt treten bei Kikura zwar weniger fehlerhafte Transformationen als bei Barabash auf, aber das Plugin stürzt auch bei fast der Hälfte der Eingabedateien ab, was darauf hindeutet, dass noch viele Spezialfälle außer Acht gelassen wurden. Der Ansatz von Barabash ist hier bedeutend stabiler, jedoch wurden in Summe mehr Fehler aufgedeckt. Da hier allerdings auch deutlich mehr Testfälle untersucht werden konnten (161 bei Kikura versus 466 bei Barabash), ist dies zu erwarten. Zusammenfassend lässt sich sagen, dass beide Ansätze offenbar viele Grenzfälle der Übersetzung derzeit noch nicht korrekt behandeln, sodass verschiedene semantische und syntaktische Probleme in der Ausgabe entstehen. Die vorliegende Implementierung transformiert dagegen alle Testfälle richtig.

Fazit

Durch die Fixture-Tests konnten sowohl die Korrektheit, als auch die Vollständigkeit der Übersetzung der Flow-Typen nach TypeScript überprüft werden. Jedoch setzt dies voraus, dass die Testfälle tatsächlich einen Großteil der möglichen Typisierung durch Flow abbilden. Obwohl die Tests mit großer Sorgfalt angelegt wurden, kann nicht ausgeschlossen werden, dass gewisse Fälle nicht bedacht wurden. In Anbetracht dessen, dass die Testfälle aber bei den zum Vergleich herangezogenen Ansätzen von Kikura und Barabash noch viele Schwachstellen offengelegt haben, kann angenommen werden, dass diese zumindest den wesentlichen Teil der Szenarien abdecken. Auch die erfolgreiche praktische Anwendung des umgesetzten Transpilers auf die Projekte von TeamShirts deutet stark darauf hin, dass sowohl die Semantik als auch die Syntax

[40] Zum Beispiel: `function f(optional?: number = 10) {}`.

der Flow-Typen korrekt nach TypeScript überführt wird. Die Äquivalenz der Transformationen ist dabei wie beschrieben nur in einzelnen Fälle nicht gegeben.

5.3.2 Semantisch äquivalente Transpilierung des Quelltexts

Eine weitere wichtige Anforderung an den Transpiler ist, dass die Semantik des ursprünglichen JavaScript-Programms durch die Übersetzung nicht verändert werden darf, damit keine neuen Programmfehler in den Quelltext eingeschleust werden. Die korrekte Übersetzung der Semantik der Flow-Typen wurde bereits im vorherigen Abschnitt behandelt. Hierbei ist anzumerken, dass die statische Typisierung aber ohnehin keinen Einfluss auf die Semantik des Programms zur *Laufzeit* hat, weil die Typen vor der Auslieferung vollständig entfernt werden. Dies setzt vorraus, dass die Programm-Transformation tatsächlich nur die Typisierung adressiert und keine unbeabsichtigte Nebenwirkung, wie zum Beispiel das Einfügen neuer Anweisungen in das Programm, hat. Ein weiterer Aspekt des umgesetzten Transcompilers, der zu abweichender Semantik führen könnte, sind die in Abschnitt 4.3.3 beschriebenen optionalen Optimierungen der Ausgabe. Durch die Option „`replace-decorators`“ werden beispielsweise Dekoratoren in bedeutungsgleiche verschachtelte Funktionsaufrufe umgewandelt. Die Einhaltung dieser Einschränkungen wurde wie ausgeführt durch die Fixture-Tests überprüft.

Ein formaler Beweis der semantischen Äquivalenz ist aufgrund des Umfangs der zwei migrierten Projekte nicht durchführbar. Jedoch kann durch Betrachtung der bestehenden Modultests die Beibehaltung der Programmwirkung nach der Übersetzung zumindest approximativ verifiziert werden. Kritisch anzumerken ist, dass dieser Ansatz nur bei einer entsprechend hohen Testabdeckung aussagekräftig ist. Während diese bei Components bei 90,5% liegt, wird bei Helios nur ein Anteil von 18,0% erreicht. Nachdem alle nicht-strikten Typfehler in beiden Projekten korrigiert worden sind, wurden die Modultests beider Projekte ausgeführt. Keiner der 373 Tests von Components und 326 Tests von Helios schlug dabei fehl, sodass die semantisch korrekten Überführung des ursprünglichen Quelltexts nach TypeScript zumindest für das Projekt Components wahrscheinlich ist. Bei der praktischen Erprobung der zwei Projekte wurde allerdings in vier Fällen ein fehlerhaftes Verhalten der Webanwendung festgestellt. Jedoch konnte jedes dieser Probleme auf eine fehlerhafte händische Korrektur neuer Typfehler zurückgeführt werden. Es wird somit angenommen, dass der Transpiler die Semantik durch die Übersetzung tatsächlich beibehält und nur wo nötig, zum Beispiel bei nicht unterstützten Flow-Typen, modifiziert.

5.3.3 Unterstützung aktueller und vorläufiger JavaScript- sowie JSX-Syntax

Die Anforderung aktuelle und vorläufige JavaScript- sowie JSX-Syntax zu unterstützen war eines der grundlegenden Kriterien, anhand derer in Abschnitt 2.3.2 verschiedene Parser, Codegeneratoren und Transpiler bezüglich ihrer Eignung als Basis für die Umsetzung des Flow-Transpilers verglichen wurden. Wie dort bereits ausgeführt ist Babel nach Kenntnisstand des Autors das einzige Werkzeug, das sowohl jegliche aktuelle als auch vorläufige JavaScript-Syntax einlesen und verarbeiten kann. Deshalb wurde der umgesetzte Transpiler auf Basis von Babel implementiert, um diese Anforderung zu erfüllen. Wie die erfolgreich durchgeführte Transpilierung der zwei Projekte von TeamShirts beweist, ist die Verarbeitung derartiger Syntax tatsächlich möglich.

Auch hier sollen die Ansätz von Kikura und Barabash bezüglich dieser Anforderung untersucht werden, um die Ergebnisse zu vergleichen. Beide Projekte wurden wie die vorliegende Implementierung als Babel-Plugin umgesetzt, sodass aktuelle ECMAScript-Syntax vollständig unterstützt wird. Darüber hinaus ermöglichen beide Projekten die Verarbeitung von JSX-Syntax und der vorläufigen Erweiterungen *Class field declarations for JavaScript* [45] und *Dynamic imports* [35]. Klassendekoratoren werden jedoch von keinem der Ansätzen unterstützt, weshalb Dateien, die derartige Syntax verwenden, nicht transpiliert werden können. Darüber hinaus wurde festgestellt, dass bestimmte korrekte Flow-Syntax in Objekttypen bei Barabash zu Laufzeitfehlern führt, weil offenbar in diesen Fällen Knoten des abstrakten Syntaxbaums fehlerhaft transformiert werden[41]. Auch bei Kikura treten bei Verwendung mancher Flow-Notationen wie zum Beispiel bei der expliziten Typumwandlung vereinzelt Syntaxfehler auf. Die vollständige Verarbeitung der Codebasis der zwei Projekte von TeamShirts wäre somit mit den Ansätzen von Kikura und Barabash nicht möglich.

5.3.4 Verarbeitung gesamter Projektverzeichnisse

Wie in Abschnitt 4.4 bereits ausführlich dargelegt wurde der Transpiler um ein Kommandozeilenprogramm erweitert, um die Anforderung der Verarbeitung gesamter Verzeichnisse zu realisieren. Die Anwendung erwartet dabei eine oder mehrere Dateien bzw. Verzeichnisse als Argument und setzt daraufhin die Übersetzung dieser Eingaben mit Hilfe des Babel-Plugins um.

[41] Vgl. Tabelle 5.6

Durch die interne Verwendung der Bibliothek *Glob* [110] können hierbei beliebig komplexe Wildcard-Muster (*glob patterns*) verarbeitet werden, um bestimmte Datei- oder Verzeichnistypen ein- und auszuschließen. Die Glob-Notation ähnelt regulären Ausdrücken in einigen Aspekten konzeptionell, sodass auch die Angabe komplizierterer Muster wie beispielsweise die Negation von Ausdrücken oder die Angabe einer Gruppe von alternativen Werten möglich ist [70]. Die in Abschnitt 3.3.4 definierten Anforderungen, dass einerseits Verzeichnisse rekursiv verarbeitet werden, andererseits die Menge der zu übersetzenden Dateien flexibel eingegrenzt werden können muss, wurden somit erfüllt.

Von den zwei zum Vergleich betrachteten Ansätzen, ermöglicht lediglich der Flow-Transpiler von Barabash die Übersetzung ganzer Projektverzeichnisse durch ein integriertes Kommandozeilenprogramm [14]. Das Babel-Plugin von Kikura bietet eine solche Funktionalität nicht.

5.3.5 Beibehaltung der Quelltextformatierung

Ergebnisse

Die letzte technische Anforderung an den Transpiler ist die möglichst originalgetreue Formatierung der Ausgabe. Weil diese bei Verwendung des Babel-Codegenerators aus den dargelegten Gründen nicht ohne Weiteres umsetzbar ist, wurde eine Formatierungsroutine auf Basis des Werkzeugs Prettier [38] implementiert. Mittels händischer Überprüfung des erzeugten TypeScript-Codes der zwei Projekte von TeamShirts, wurden inkorrekt formatierte Dateien ermittelt. Hierunter werden Dateien verstanden, in denen Ausdrücke, mit Ausnahme von Objektliteralen, anders als im ursprünglichen Quelltext umgebrochen oder Leerzeilen und Kommentare falsch platziert werden. Sobald eines dieser Kriterien vorliegt, wird die Formatierung als fehlerhaft betrachtet.

Tabelle 5.7 zeigt die Ergebnisse dieser manuellen Überprüfung. Dabei werden die erreichten Werte der vorliegende Implementierung dem Ansatz von Barabash [14] gegenüber gestellt, um die Zahlen besser einordnen zu können. Das Babel-Plugin von Kikura [87] bietet keine Möglichkeit, die Ausgabe zu formatieren und wird daher nicht in den Vergleich miteinbezogen. Wie die Tabelle zeigt, schlägt das Verfahren sowohl in der vorliegenden Umsetzung als auch bei Barabash in einigen Fällen fehl. Jedoch erzielt die in dieser Arbeit implementierte For-

matierungsroutine insgesamt bessere Ergebnisse: Während hier in Summe 32 Dateien (4,7%) fehlerhaft formatiert werden, liegt der Anteil bei Barabash bei 48 Dateien (7,0%).

Tabelle 5.7: Anteil fehlerhaft formatierter Ausgabedateien in den Projekten Components und Helios im Vergleich zu Ansatz von Barabash [14].

		VORL. IMPL.		BARABASH	
Projekt	**Dateien**	**fehlerhaft**	**rel. Anteil**	**fehlerhaft**	**rel. Anteil**
Components	331	11	3,5%	31	9,4%
Helios	353	21	5,9%	17	4,8%
Insgesamt	684	32	4,7%	48	7,0%

Fehleranalyse

Zur Feststellung der Ursachen der fehlerhaften Formatierung durch den in dieser Arbeit umgesetzten Transpiler, wurde die Verarbeitung der inkorrekt ausgegebenen Dateien untersucht. Weil das Verfahren zeilenbasiert arbeitet, ist die Konsistenz der Umbrüche für das Gelingen der Formatierung entscheidend, da es andernfalls zu einem Versatz der betrachteten Eingabe- und Ausgabgezeilen kommt. Die Betrachtung der Originaldateien, welche fehlerhaft formatiert wurden, hat gezeigt, dass dies in Folge von bestimmter, eher selten vorkommender Quelltextformatierung auftritt. Nachfolgend soll das Problem anhand eines Beispiel für einen solchen Fall veranschaulicht werden:

```
1 if (
2   a &&
3   b &&    // Kommentar, der Zweck von 'b' beschreibt
4   c === 0 // Kommentar, der Zweck von 'c' beschreibt
5 ) {
6   // ...
7 }
```

Quelltext 5.3: Beispiel für Formatierung der Eingabe, die falsch in die Ausgabe übernommen wird.

Der boolesche Ausdruck, der als Argument der bedingten Anweisung in Zeile 1 dient, wird mehrmals umgebrochen, damit der Zweck der einzelnen Elemente durch einen Zeilenkommentar erläutert werden kann. Der umgesetzte Flow-Transpiler ignoriert Kommentare der Eingabe bei Generierung der TypeScript-Ausgabe generell, weil Babel diese wie ausgeführt ohnehin zum Teil falsch platziert. Erst durch die Formatierungsroutine werden die Kommentare im

Nachgang wieder eingefügt, um deren korrekte Positionierung zu erzielen. Diese schlägt für das Beispiel fehl, weil der boolesche Ausdruck in der erzeugten Ausgabe aufgrund der entfernten Kommentare nicht länger umgebrochen wird (vgl. Quelltext 5.4). Da der Zeilenumbruch in der Ein- und Ausgabe in diesem Fall somit inkonsistent ist, scheitert die Übertragung der Formatierung beim Vergleich dieser und aller nachfolgenden Zeilen.

```
1 if (foo && bar && baz === 0) {
2   // ...
3 }
```

Quelltext 5.4: Generierte TypeScript-Ausgabe der Eingabe in 5.3 *vor* Ausführung der Formatierungsroutine.

Wie die weitere Fehleranalyse gezeigt hat, ergibt sich bei der Auflistung der formalen Parameter von Funktionen die gleiche Problematik, wenn diese analog zum Beispiel umgebrochen werden. Zur Lösung dieser fehlerhaften Fälle sollte die angepasste Version von Prettier so erweitert werden, dass auch das Argument bedingter Anweisungen und die Auflistung von Parametern stets umgebrochen werden.

Fazit

Zwar konnte die originalgetreue Formatierung nicht vollständig umgesetzt werden, weil nicht alle Ausgabedateien korrekt formatiert werden, dennoch wurde der Programmierstil in 95,3% der Fälle akkurat beibehalten. Da die manuelle Korrektur der insgesamt 32 fehlerhaften Dateien mit geringem Zeitaufwand von wenigen Stunden durchführbar ist, kann die geforderte Funktion damit als größtenteils erfüllt betrachtet werden.

6 Schlussbetrachtung

6.1 Zusammenfassung der Ergebnisse

In der vorliegenden Masterarbeit wurde die Fragestellung behandelt, wie das statische Typsystem Flow [24] nach TypeScript [125] übersetzt werden kann, um eine derartige Migration von JavaScript-Quelltexten zu automatisieren. Hierfür wurde ein Transpiler auf Grundlage von Babel entwickelt, mit dessen Hilfe zwei Projekte des Unternehmens Spreadshirt erfolgreich nach TypeScript übertragen wurden.

Zu Beginn wurden verschiedene technische Anforderungen an den Transpiler erarbeitet, die dessen Funktionsumfang festlegen: Dieser soll in der Lage sein, jegliche Flow-Syntax korrekt in äquivalente TypeScript-Ausdrücke zu übersetzen. Die Äquivalenz der Übersetzungen wurde größtenteils umgesetzt, jedoch existieren, wie beschrieben, einerseits Typen, die grundsätzlich nicht bedeutungsgleich nach TypeScript überführt werden können, andererseits konnte für drei Hilfstypen von Flow kein äquivalenter TypeScript-Ausdruck gefunden werden. Die Erzielung sowohl der Vollständigkeit als auch der Korrektheit der Transformationen wurde durch Fixture-Tests überprüft. Weiterhin wurde vorgegeben, dass die Ausführung des Transpilers die Semantik des Eingabeprogramms nicht verändern darf, damit keine neuen Programmfehler in die Ausgabe eingeführt werden. Mit Hilfe der bestehenden Modultests der zwei Projekte wurde näherungsweise gezeigt, dass der Transcompiler die Semantik des Programms tatsächlich nur wo nötig verändert. Auf Grundlage der Rahmenbedingungen innerhalb des Unternehmens wurde außerdem spezifiziert, dass der Transpiler aktuelle JavaScript-Syntax gemäß des ECMAScript-Standards 2019 [42], bestimmte vorläufige Spracherweiterungen wie zum Beispiel Klassendekoratoren und JSX-Syntax unterstützen muss. Durch den Einsatz von Babel und verschiedenen vorgegebenen Plugins wurde diese Anforderung vollständig erreicht. Zuletzt muss der Transpiler zum einen gesamte Projektverzeichnisse verarbeiten können, zum anderen eine originalgetreue Formatierung der Ausgabe ermöglichen. Beide Vorgaben wurden durch die Umsetzung eines Kommandozeilenprogramms, das die geforderte Funktionen implementiert, erfüllt. Allerdings konnte gezeigt werden, dass die Formatierung bei den zwei migrierten Projekten nur für 95,3% der Dateien präzise beibehalten wurde.

Mit dem Wechsel des statischen Typsystems wurden verschiedene Ziele verfolgt, die in Absprache mit dem Unternehmen definiert wurden: So sollten Typ- und Programmfehler im bestehenden Programmcode aufgedeckt und deren Erkennung in Zukunft vereinfacht werden. Nach der Migration wurden in beiden Projekten zahlreiche Typverletzungen durch TypeScript erkannt. Der Großteil dieser neu aufgetretenen Fehler kann auf die ausgeführten grundlegenden Unterschiede von Flow und TypeScript, die Integration externer Typdeklarationen für Bibliotheken und auf Unzulänglichkeiten der vorherigen Flow-Typisierung zurückgeführt werden. Durch einen kleinen Teil der Typverletzungen wurden aber tatsächlich bisher unerkannte unkritische Programmfehler offengelegt. Weiterhin wurde durch die Migration angestrebt, die Unterstützung für externe Softwarebibliotheken durch das Typsystem zu verbessern. Diese Zielsetzung wurde erreicht: Wie gezeigt bestehen bei TypeScript Typdefinitionen für etwa zehn mal mehr Bibliotheken als bei Flow. Jedoch konnten bei der konkreten Betrachtung einer Auswahl der meistverwendesten Pakete in den zwei Projekten von Spreadshirt nur geringfügige Vorteile bei TypeScript hinsichlicht der Aktualität und der Qualität der Typdeklarationen festgestellt werden. Das dritte Ziel, die Steigerung der Performance der Typüberprüfungen, wurde nur teilweise erreicht. Nur in Einzelfällen konnte im Projekt Helios bei TypeScript eine geringere Laufzeit der vollständigen Berechnungen der Typkorrektheit gegenüber Flow gemessen werden. Aufgrund von technischen Einschränkungen, welche die Messung verfälscht hätten, konnte die Dauer von inkrementellen Typüberprüfungen in dieser Arbeit nicht untersucht werden. Grundsätzlich scheint die stark parallelisierte Architektur von Flow der von TypeScript bezüglich der Performance überlegen. Zuletzt wurden durch den Wechsel zu TypeScript Vorteile hinsichtlich der Zukunftssicherheit der Technologie erwartet. In der Tat wird TypeScript im Vergleich zu Flow offener entwickelt, gemeldete Programmfehler werden im Allgemeinen schneller bearbeitet und die Kommunikation ist weitaus transparenter. Da bei TypeScript regelmäßig detaillierte Projektpläne veröffentlicht werden, besteht hier eine insgesamt größere Zukunftssicherheit bezüglich der Weiterentwicklung des Systems.

Zur Einordnung der Ergebnisse wurden die Ansätze von Kikura [87] und Barabash [14] zur Transpilierung von Flow nach TypeScript herangezogen. Dabei konnte nachgewiesen werden, dass bei Kikura und Barabash 54,8% bzw. 9,7% der Fixture-Dateien zur Erprobung der korrekten Übersetzung aufgrund von Laufzeitfehlern überhaupt nicht verarbeitet werden können und in 1,9% bzw. 11,0% der anderen Testfälle nicht äquivalente Transformationen auftreten. Weiterhin wurde aufgezeigt, dass beide Ansätze auch bei bestimmter valider Flow-Syntax infolge von Programmfehlern abstürzen. Barabash ermöglicht wie der umgesetzte Transpiler die Verarbeitung ganzer Projektverzeichnisse. Allerdings wurden hier schlechtere Ergebnisse hinsichtlich der Formatierung der Ausgabe festgestellt (4,7% bzw. 7,0% Fehlerrate).

Zusammenfassend kann die Migration zu TypeScript bei Spreadshirt als erfolgreich betrachtet werden: Wie ausgeführt wurden einerseits die gesetzten Ziele größtenteils erreicht, andererseits die technischen Anforderungen an den Transpiler mehrheitlich erfüllt. Durch den Übersetzer konnte der Arbeitsaufwand und die Fehleranfälligkeit des Wechsels des Typsystems deutlich reduziert werden. Die auf diese Weise migrierten Projekte Components und Helios wurden inzwischen in einer neuen Version ausgeliefert und sind produktiv im Einsatz. Auch das Entwicklerteam von TeamShirts ist laut eigener Aussage sehr zufrieden mit dem vollzogenen Wechsel zu TypeScript.

6.2 Mögliche Erweiterungen des Projekts

Zuletzt soll auf zwei Aspekte dieser Arbeit eingegangen werden, die zukünftig optimiert werden könnten: So treten bei der originalgetreuen Formatierung der Ausgabe durch den Transpiler wie gezeigt teilweise noch Fehler auf. Das implementierte zeilenbasierte Verfahren funktioniert zwar in den allermeisten Fällen akkurat, ist aber nicht besonders robust. Sobald eine Abweichung beim Vergleich der Ein- und Ausgabe auftritt, ist eine korrekte Behandlung der verbleibenden Datei unmöglich[42]. Ein aussichtsreicher Ansatz wäre, die gesamte Quelltexttransformation durch einen konkreten statt abstrakten Syntaxbaum umzusetzen, damit die Information über die Formatierung des ursprünglichen Quelltexts präzise beibehalten wird. Bereits während der Entwicklung des Transpilers wurde dieser Ansatz in Betracht gezogen und nach geeigneten Bibliotheken gesucht. Dabei wurde das Projekt *CST* (*Concrete Syntax Tree*) [133] gefunden, welches das Einlesen konkreter Syntaxbäume für JavaScript ermöglicht. Allerdings werden hierbei weder Flow, noch TypeScript unterstützt, sodass die angestrebte Verwendung so nicht möglich ist. Falls das Projekt aber entsprechend erweitert werden würde, könnte die Programmtransformation in Zukunft auf diese Weise durchgeführt werden.

Aufgrund des begrenzten zeitlichen Rahmens dieser Masterarbeit konnten in den zwei migrierten Projekten von TeamShirts nur die nicht-strikte Überprüfung von Typfehlern durch TypeScript realisiert werden. Zur weiteren Steigerung der statischen Typsicherheit sollte deshalb zukünftig in beiden Projekten strikte Überprüfungen aktiviert und die resultierenden Typfehler daraufhin korrigiert werden.

[42] Vgl. Abschnitt 5.3.5.

Quellenverzeichnis

[1] *[RFC] Migrate Jest to TypeScript*. Issue #7554. Sep. 2017. URL: https://github.com/facebook/jest/pull/7554 (besucht am 05. 11. 2019) (siehe S. 74).

[2] Dan Abramov. *React Components, Elements, and Instances*. 18. Dez. 2015. URL: https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html (besucht am 07. 11. 2019) (siehe S. 65).

[3] Khan Academy. *About | Khan Academy*. 2019. URL: https://www.khanacademy.org/about (besucht am 27. 09. 2019) (siehe S. 31).

[4] sprd.net AG. *Über uns*. 2019. URL: https://www.teamshirts.de/ueber-uns (besucht am 06. 10. 2019) (siehe S. 29).

[5] sprd.net AG. *Unternehmen*. 2019. URL: https://www.spreadshirt.de/unternehmen-C2410 (besucht am 16. 08. 2019) (siehe S. 29).

[6] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811 (siehe S. 20 f., 23).

[7] Roberto M. Amadio und Luca Cardelli. „Subtyping Recursive Types“. In: *ACM Trans. Program. Lang. Syst.* 15.4 (Sep. 1993), S. 575–631. ISSN: 0164-0925. DOI: 10.1145/155183.155231 (siehe S. 7).

[8] Christopher Anderson und Paola Giannini. „Type checking for JavaScript“. In: *Electronic Notes in Theoretical Computer Science* 138.2 (2005), S. 37–58 (siehe S. 2).

[9] *Apache License, Version 2.0*. The Apache Software Foundation, Jan. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0 (besucht am 21. 10. 2019) (siehe S. 16).

[10] Andrew W. Appel und Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd. New York, NY, USA: Cambridge University Press, 2003. ISBN: 052182060X (siehe S. 20).

[15] Kent Beck. *Extreme Programming Explained: Embrace Change*. The XP Series. Reading, Mass. [u.a.]: Addison-Wesley, 2000. ISBN: 9780201616415 (siehe S. 39).

[16] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003 (siehe S. 39).

[17] Gavin Bierman, Martin Abadi und Mads Torgersen. „Understanding TypeScript“. In: *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, S. 257–281. ISBN: 978-3-662-44201-2. DOI: 10.1007/978-3-662-44202-9_11 (siehe S. 3, 7, 16 ff.).

[19] Stephan Boyer. *What are covariance and contravariance?* Juli 2017. URL: https://www.stephanboyer.com/post/132/what-are-covariance-and-contravariance (besucht am 25. 10. 2019) (siehe S. 7).

[20] Luca Cardelli. „Type systems“. In: *ACM Computing Surveys* 28.1 (1996), S. 263–264 (siehe S. 2 f., 6 f.).

[21] 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).

[22] 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).

[23] Avik Chaudhuri. *What the Flow team has been up to*. 28. Jan. 2019. URL: https://medium.com/flow-type/what-the-flow-team-has-been-up-to-54239c62004f (besucht am 04. 11. 2019) (siehe S. 33, 74).

[24] Avik Chaudhuri et al. „Fast and Precise Type Checking for JavaScript“. In: *Proc. ACM Program. Lang.* 1.48 (Okt. 2017), 48:1–48:30. ISSN: 2475-1421. DOI: 10.1145/3133872. URL: http://doi.acm.org/10.1145/3133872 (siehe S. b, 2 f., 7–10, 17 f., 33, 86).

[26] Microsoft Corporation. *Compiler Options*. 2019. URL: https://www.typescriptlang.org/docs/handbook/compiler-options.html (besucht am 23. 10. 2019) (siehe S. 17, 62, 66).

[27] Microsoft Corporation. *GitHub API v3 | GitHub Developer Guide*. 2019. URL: https://developer.github.com/v3/ (besucht am 05. 11. 2019) (siehe S. 74).

[29] Microsoft Corporation. *TypeScript Design Goals*. Sep. 2014. URL: https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals (besucht am 21. 10. 2019) (siehe S. 17 f.).

[30] Microsoft Corporation. *var (C# Reference)*. Juli 2015. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/var (besucht am 19. 10. 2019) (siehe S. 15).

[31] Douglas Crockford. *JavaScript: The World's Most Misunderstood Programming Language*. 2001. URL: https://crockford.com/javascript/javascript.html (besucht am 16. 07. 2019) (siehe S. 1).

[32] Douglas Crockford. *The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language*. März 2008. URL: http://crockford.com/javascript/popular.html (besucht am 22. 07. 2019) (siehe S. 1).

[33] Jamie Davis. *Don't Block the Event Loop (or the Worker Pool)*. Okt. 2019. URL: https://nodejs.org/ru/docs/guides/dont-block-the-event-loop/ (besucht am 29. 10. 2019) (siehe S. 71).

[35] Domenic Denicola. *import()*. Stage 4. Ecma International, Technical Committee 39, 2019. URL: https://github.com/tc39/proposal-dynamic-import (besucht am 02. 08. 2019) (siehe S. 42, 82).

[36] MDN Web Docs. *Default parameters*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters (besucht am 31. 10. 2019) (siehe S. 80).

[37] MDN Web Docs. *Web Components*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components (besucht am 16. 08. 2019) (siehe S. 29).

[39] Ecma International. *TC39 - ECMAScript*. 2019. URL: https://www.ecma-international.org/memento/tc39.htm (besucht am 02. 08. 2019) (siehe S. 23).

[40] Ecma International. *The TC39 Process*. 2019. URL: https://tc39.es/process-document/ (besucht am 02. 08. 2019) (siehe S. 23).

[41] *ECMAScript 2017 Language Specification*. 8th. Ecma International. Geneva, Juni 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf (besucht am 30. 09. 2019) (siehe S. 35).

[42] *ECMAScript 2019 Language Specification*. 10th. Ecma International. Geneva, Juni 2019. URL: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf (besucht am 23. 07. 2019) (siehe S. 2, 15, 23, 35, 53, 86).

[43] *ECMAScript: A general purpose, cross-platform programming language*. 1st. Ecma International. Geneva, Juni 1997. URL: https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf (besucht am 03. 08. 2019) (siehe S. 1).

[44] Daniel Ehrenberg. *JavaScript Decorators*. Stage 2. Ecma International, Technical Committee 39, 2019. URL: https://github.com/tc39/proposal-decorators (besucht am 02. 08. 2019) (siehe S. 42, 54).

[45] Daniel Ehrenberg und Jeff Morrison. *Class field declarations for JavaScript*. Stage 3. Ecma International, Technical Committee 39, 2019. URL: https://github.com/tc39/proposal-class-fields (besucht am 02. 08. 2019) (siehe S. 35, 42, 82).

[46] *Existential type?* Issue #14466. März 2017. URL: https://github.com/Microsoft/TypeScript/issues/14466 (besucht am 15. 07. 2019) (siehe S. 76).

[47] Facebook Inc. *Creating Library Definitions*. Flow. 2019. URL: https://flow.org/en/docs/libdefs/creation/ (besucht am 18. 10. 2019) (siehe S. 16).

[49] Facebook Inc. *Generic Types*. Flow. 2019. URL: https://flow.org/en/docs/types/generics/ (besucht am 21. 10. 2019) (siehe S. 20).

[50] Facebook Inc. *Handling Events*. 2019. URL: https://reactjs.org/docs/handling-events.html (besucht am 29. 09. 2019) (siehe S. 35).

[51] Facebook Inc. *Installation*. 2019. URL: https://flow.org/en/docs/install/ (besucht am 16. 10. 2019) (siehe S. 8).

[52] Facebook Inc. *Introducing JSX*. 2019. URL: https://reactjs.org/docs/introducing-jsx.html (besucht am 02. 08. 2019) (siehe S. 24, 35).

[54] Facebook Inc. *Lint Rule Reference*. 2018. URL: https://flow.org/en/docs/linting/rule-reference/#toc-deprecated-type (besucht am 11. 08. 2019) (siehe S. 52, 76).

[55] Facebook Inc. *Maybe Types*. Flow. 2019. URL: https://flow.org/en/docs/types/maybe/ (besucht am 25. 10. 2019) (siehe S. 49).

[56] Facebook Inc. *Nominal & Structural Typing*. Flow. 2019. URL: https://flow.org/en/docs/lang/nominal-structural/ (besucht am 17. 10. 2019) (siehe S. 10).

[58] Facebook Inc. *Type Annotation*. Flow. 2019. URL: https://flow.org/en/docs/types/ (besucht am 15. 07. 2019) (siehe S. 12 ff., 77).

[59] Facebook Inc. *Type System*. Flow. 2019. URL: https://flow.org/en/docs/lang/ (besucht am 16. 10. 2019) (siehe S. 9).

[60] Facebook Inc. *Type Variance*. Flow. 2019. URL: https://flow.org/en/docs/lang/variance/ (besucht am 25. 10. 2019) (siehe S. 7).

[61] Facebook Inc. *Types & Expressions*. Flow. 2019. URL: https://flow.org/en/docs/lang/types-and-expressions/ (besucht am 16. 10. 2019) (siehe S. 6, 9).

[62] Facebook Inc. *Utility Types*. Flow. 2019. URL: https://flow.org/en/docs/types/utilities/ (besucht am 15. 07. 2019) (siehe S. 14 f.).

[63] Facebook Inc. *Web Components*. 2019. URL: https://reactjs.org/docs/web-components.html (besucht am 16. 08. 2019) (siehe S. 29).

[64] Facebook Inc. *Width Subtyping*. Flow. 2019. URL: https://flow.org/en/docs/lang/width-subtyping/ (besucht am 18. 10. 2019) (siehe S. 13, 65).

[65] *Facebook's own Flow adoption?* Issue #7365. Jan. 2017. URL: https://github.com/facebook/flow/issues/7365 (besucht am 05. 11. 2019) (siehe S. 73).

[67] Martin Fowler. *Transparent Compilation*. Feb. 2013. URL: https://martinfowler.com/bliki/TransparentCompilation.html (besucht am 21. 10. 2019) (siehe S. 23).

[68] Elisabeth Freeman et al. *Head First Design Patterns*. 2nd. O'Reilly & Associates, Inc., 2004. ISBN: 978-0-5960-07126 (siehe S. 26).

[69] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Aufl. Addison-Wesley Professional, 1994. ISBN: 0201633612 (siehe S. 26).

[70] *GLOB(3) – Linux Programmer's Manual*. Version 7.6. The Linux man-pages project. 6. März 2019. URL: http://man7.org/linux/man-pages/man3/glob.3.html (besucht am 23. 10. 2019) (siehe S. 83).

[72] Jesset Hallett. *Advanced features in Flow*. Mai 2015. URL: http://sitr.us/2015/05/31/advanced-features-in-flow.html (besucht am 19. 10. 2019) (siehe S. 15).

[74] Anders Heljsberg. *aheljsberg (Anders Heljsberg*. 2019. URL: https://github.com/ahejlsberg (besucht am 21. 10. 2019) (siehe S. 16).

[75] Dave Herman et al. *The ESTree Spec*. Juli 2019. URL: https://github.com/estree/estree (besucht am 05. 08. 2019) (siehe S. 22, 24, 28).

[78] Evgeniy Ilyushin und Dmitry Namiot. „On source-to-source compilers". In: *International Journal of Open Information Technologies* 4 (Apr. 2016), S. 48–51 (siehe S. 4, 20 f.).

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

[80] *Interfaces · TypeScript*. Microsoft Corporation. 2019. URL: https://www.typescriptlang.org/docs/handbook/interfaces.html (besucht am 25. 10. 2019) (siehe S. 76).

[81] ISO. *ISO C Standard 1999*. Techn. Ber. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (besucht am 02. 10. 2019) (siehe S. 6).

[82] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sep. 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372 (siehe S. 15).

[83] Ron Jeffries. *What is Extreme Programming?* März 2011. URL: https://ronjeffries.com/xprog/what-is-extreme-programming/ (besucht am 15. 07. 2019) (siehe S. 39).

[84] *JSX · TypeScript*. Microsoft Corporation. 2019. URL: https://www.typescriptlang.org/docs/handbook/jsx.html (besucht am 20. 07. 2019) (siehe S. 58).

[86] Brian W. Kernighan und P. J. Plauger. *The Elements of Programming Style*. 2nd. New York, NY, USA: McGraw-Hill, Inc., 1982. ISBN: 0070342075 (siehe S. 36).

[89] Jamie Kyle, Sebastian McKenzie, Henry Zhu et al. *Babel Handbook*. Version 7.6. Babel. Sep. 2017. URL: https://github.com/jamiebuilds/babel-handbook/blob/master/translations/en/user-handbook.md (besucht am 11. 07. 2019) (siehe S. 25, 27 f., 42).

[91] Donna Malayeri und Jonathan Aldrich. „Integrating Nominal and Structural Subtyping". In: *ECOOP 2008 – Object-Oriented Programming*. Hrsg. von Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 260–284. ISBN: 978-3-540-70592-5 (siehe S. 7).

[92] Sebastian McKenzie. *@babel/parser (Babylon) AST node types*. Version 7.6. Babel. Juli 2019. URL: https://github.com/babel/babel/blob/master/packages/babel-parser/ast/spec.md (besucht am 17. 07. 2019) (siehe S. 28, 34, 45, 77).

[94] Microsoft Corporation. *Architectural Overview*. TypeScript. Feb. 2017. URL: https://github.com/microsoft/TypeScript/wiki/Architectural-Overview (besucht am 21. 10. 2019) (siehe S. 18 f.).

[95] Microsoft Corporation. *Standalone Server (tsserver)*. TypeScript. Okt. 2019. URL: https://github.com/microsoft/TypeScript/wiki/Standalone-Server-%28tsserver%29 (besucht am 22. 10. 2019) (siehe S. 19).

[96] Jason Miller und Addy Osmani. *Rendering on the Web*. Feb. 2019. URL: https://developers.google.com/web/updates/2019/02/rendering-on-the-web (besucht am 27. 08. 2019) (siehe S. 30).

[97] *MIT License*. Massachusetts Institute of Technology, 2019. URL: https://opensource.org/licenses/MIT (besucht am 01. 07. 2019) (siehe S. 8, XVI).

[98] John C Mitchell und Krzysztof Apt. *Concepts in programming languages*. Cambridge University Press, 2003 (siehe S. 3).

[99] *Multicore compilation on roadmap?* Issue #12987. Dez. 2016. URL: https://github.com/Microsoft/TypeScript/issues/12987 (besucht am 28. 10. 2019) (siehe S. 70).

[101] Dor Nir, Shmuel Tyszberowicz und Amiram Yehudai. „Locating Regression Bugs“. In: *Hardware and Software: Verification and Testing*. Hrsg. von Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 218–234. ISBN: 978-3-540-77966-7 (siehe S. 39).

[102] OGM. *OMG Unified Modeling Language (OMG UML)*. 2.5. Object Management Group. Aug. 2017. URL: https://www.omg.org/spec/UML/2.5.1 (besucht am 16. 07. 2019) (siehe S. 42).

[103] Michael Olan. „Unit testing: test early, test often“. In: *Journal of Computing Sciences in Colleges* 19.2 (2003), S. 319–328 (siehe S. 40).

[104] L. D. Paulson. „Developers shift to dynamic programming languages“. In: *Computer* 40.2 (Feb. 2007), S. 12–15. ISSN: 0018-9162. DOI: 10.1109/MC.2007.53 (siehe S. 1).

[105] Michael Pradel und Koushik Sen. „The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript“. In: *ECOOP*. 2015 (siehe S. 1).

[106] Gregor Richards et al. „An Analysis of the Dynamic Behavior of JavaScript Programs“. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, S. 1–12. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806598. URL: http://doi.acm.org/10.1145/1806596.1806598 (siehe S. 2).

[107] *Roadmap*. Nov. 2019. URL: https://github.com/microsoft/TypeScript/wiki/Roadmap (besucht am 05. 11. 2019) (siehe S. 74).

[108] *Roadmap/Path to V1?* Issue #4785. Sep. 2017. URL: https://github.com/microsoft/TypeScript/issues/11588 (besucht am 05. 11. 2019) (siehe S. 73).

[109] Daniel Rosenwasser. *TypeScript and Babel 7*. 27. Aug. 2018. URL: https://devblogs.microsoft.com/typescript/typescript-and-babel-7/ (besucht am 21. 10. 2019) (siehe S. 17).

[111] Ulrich Schöpp. *Praktikum Compilerbau – Wintersemester 2014/15*. Ludwig-Maximilians-Universität München, Institut für Informatik, Jan. 2019. URL: https://www.tcs.ifi.lmu.de/lehre/ws-2014-15/compilerbau/material/folien (besucht am 07. 10. 2019) (siehe S. 21 f.).

[112] *Semantic Versioning 2.0.0*. Version 2.0. Preston-Werner, Tom. 2013. URL: https://semver.org/lang/de/ (besucht am 04. 11. 2019) (siehe S. 67).

[113] 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).

[114] Jeremy Siek und Walid Taha. „Gradual Typing for Objects“. In: Aug. 2007, S. 2–27. DOI: 10.1007/978-3-540-73589-2_2 (siehe S. 18).

[115] Chris Smith. *What To Know Before Debating Type Systems*. 2008. URL: https://cdsmith.wordpress.com/2011/01/09/an-old-article-i-wrote/ (besucht am 02. 10. 2019) (siehe S. 3).

[116] CACM Staff. „React: Facebook's functional turn on writing Javascript“. In: *Communications of the ACM* 59.12 (2016), S. 56–62 (siehe S. 29).

[117] *Support multi-threaded compilation for –build*. Issue #30235. März 2019. URL: https://github.com/microsoft/TypeScript/issues/30235 (besucht am 29. 10. 2019) (siehe S. 73).

[118] *Support some non-structural (nominal) type matching*. Issue #202. Juli 2014. URL: https://github.com/Microsoft/TypeScript/issues/202 (besucht am 25. 10. 2019) (siehe S. 34, 49, 76).

[120] Nikhil Swamy et al. „Gradual typing embedded securely in JavaScript“. In: *ACM SIGPLAN Notices*. Bd. 49. 1. ACM. 2014, S. 425–437 (siehe S. 1 ff.).

[121] 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).

[122] *The world's leading software development platform · GitHub*. 2019. URL: https://github.com/ (besucht am 14. 10. 2019) (siehe S. 24, 67).

[123] Linda Torczon und Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012088478X (siehe S. 20 ff.).

[124] *TS1093: Can't specify return types on constructors*. Issue #11588. Okt. 2016. URL: https://github.com/microsoft/TypeScript/issues/11588 (besucht am 25. 10. 2019) (siehe S. 76).

[125] *TypeScript – Language Specification*. 1.8. Microsoft Corporation. Jan. 2016. URL: https://github.com/microsoft/TypeScript/raw/master/doc/TypeScript%20Language%20Specification.pdf (besucht am 25. 06. 2019) (siehe S. b, 3, 7 f., 16 f., 33, 86).

[126] Johannes Waldmann. *Prinzipien von Programmiersprachen – Vorlesung Wintersemester 2007 – 2018*. Hochschule für Technik, Wirtschaft und Kultur Leipzig, Jan. 2019. URL: https://www.imn.htwk-leipzig.de/~waldmann/edu/ws18/pps/folien/skript.pdf (besucht am 31. 07. 2019) (siehe S. 2 f., 21).

[127] *What is Babel?* Version 7.6. Babel. Okt. 2019. URL: https://babeljs.io/docs/en/ (besucht am 20. 10. 2019) (siehe S. 25).

[128] Kirsten Winter et al. „Path-sensitive data flow analysis simplified“. In: *International Conference on Formal Engineering Methods*. Springer, 2013, S. 415–430 (siehe S. 9).

[129] A.K. Wright und M. Felleisen. „A Syntactic Approach to Type Soundness“. In: *Information and Computation* 115.1 (1994), S. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL: http://www.sciencedirect.com/science/article/pii/S0890540184710935 (siehe S. 6).

[131] Shams Zakhou. *The Dart type system*. Sep. 2019. URL: https://dart.dev/guides/language/sound-dart (besucht am 16. 10. 2019) (siehe S. 6).

[132] Henry Zhu. *The State of Babel*. Dez. 2016. URL: https://babeljs.io/blog/2016/12/07/the-state-of-babel (besucht am 22. 07. 2019) (siehe S. 28).

Software

[11] Babel. *@babel/generator*. Version 7.6. 2019. URL: https://babeljs.io/docs/en/babel-generator (besucht am 02. 08. 2019) (siehe S. 44, 59).

[12] Babel. *@babel/parser*. Version 7.6. Babylon. 2019. URL: https://babeljs.io/docs/en/babel-parser (besucht am 05. 08. 2019) (siehe S. 25, 28).

[13] Babel. *@babel/types*. Version 7.6. 2019. URL: https://babeljs.io/docs/en/babel-types (besucht am 07. 08. 2019) (siehe S. 28, 45, 47, 77).

[14] Kevin Barabash. *flow-to-ts*. Version 0.1. 2019. URL: https://github.com/Khan/flow-to-ts (besucht am 27. 08. 2019) (siehe S. 31, 61, 79, 83 f., 87).

[18] David Bonnet. *Astring - Tiny and fast JavaScript code generator from an ESTree-compliant AST*. Version 1.3. 2015. URL: https://babeljs.io/ (besucht am 11. 10. 2019) (siehe S. 24).

[25] Boris Cherny. *Flow-to-TypeScript*. 2019. URL: https://github.com/bcherny/flow-to-typescript (besucht am 27. 08. 2019) (siehe S. 30).

[28] Microsoft Corporation. *TypeScript*. 2019. URL: https://github.com/microsoft/TypeScript/ (besucht am 21. 10. 2019) (siehe S. 16, 63).

[34] *DefinitelyTyped - The repository for high quality TypeScript type definitions*. 2019. URL: https://github.com/DefinitelyTyped/DefinitelyTyped (besucht am 03. 11. 2019) (siehe S. 66).

[38] Lucas Duailibe. *Prettier · Opinionated Code Formatter*. Version 1.18. 2019. URL: https://prettier.io/ (besucht am 02. 08. 2019) (siehe S. 59 f., 83, XVI).

[48] Facebook Inc. *Flow*. 2019. URL: https://github.com/facebook/flow (besucht am 21. 10. 2019) (siehe S. 8).

[53] Facebook Inc. *Jest - Delightful JavaScript Testing*. Version 24.8. 2019. URL: https://jestjs.io/ (besucht am 30. 07. 2019) (siehe S. 40, 74).

[57] Facebook Inc. *React - A JavaScript library for building user interfaces*. Version 16.8. 2019. URL: https://reactjs.org (besucht am 22. 03. 2019) (siehe S. 29, 32).

[66] *flow-typed/flow-typed: A central repository for Flow library definitions*. 2019. URL: https://github.com/flow-typed/flow-typed (besucht am 03. 11. 2019) (siehe S. 66).

[71] Jonathan Gruber. *Reflow: Babel plugin to transpile a Flow typed codebase to TypeScript*. 2019. URL: https://github.com/grubersjoe/reflow (besucht am 31. 07. 2019) (siehe S. 44, 55).

[73] Marijn Haverbeke. *Acorn: yet another JavaScript parser*. Version 7.1. 2012. URL: http://marijnhaverbeke.nl/blog/acorn.html (besucht am 11. 10. 2019) (siehe S. 24).

[76] Ariya Hidayat. *ECMAScript parsing infrastructure for multipurpose analysis*. Version 4.0. 2011. URL: https://esprima.org (besucht am 11. 10. 2019) (siehe S. 24).

[77] Heikki Hokkanen. *GitStats - git history statistics generator*. 2015. URL: http://gitstats.sourceforge.net/ (besucht am 03. 11. 2019) (siehe S. 67).

[85] David Keppel, David MacKenzie und Assaf Gordon. *GNU Time*. Version 2019. 2019. URL: https://www.gnu.org/software/time/ (besucht am 27. 10. 2019) (siehe S. 69).

[87] Yuichiro Kikura. *babel-plugin-flow-to-typescript*. Version 0.3. 2019. URL: https://github.com/Kiikurage/babel-plugin-flow-to-typescript (besucht am 27. 08. 2019) (siehe S. 30, 61, 79, 83, 87).

[88] Tobias Koppers. *webpack*. 2019. URL: https://webpack.js.org/ (besucht am 07. 11. 2019) (siehe S. 64).

[90] Robert M. Love. *taskset - set or retrieve a process's CPU affinity*. Version 2.34. 2019. URL: http://man7.org/linux/man-pages/man1/taskset.1.html (besucht am 28. 10. 2019) (siehe S. 71).

[93] Sebastian McKenzie et al. *Babel - The compiler for next generation JavaScript*. Version 7.6. 2018. URL: https://babeljs.io/ (besucht am 22. 03. 2019) (siehe S. b, 8, 17, 23 f., 27, 37).

[100] Ben Newman. *recast - JavaScript syntax tree transformer, nondestructive pretty-printer, and automatic source map generator*. Version 0.18. 2012. URL: https://github.com/benjamn/recast (besucht am 11. 10. 2019) (siehe S. 24 f.).

[110] Isaac Z. Schlueter. *Glob*. Version 7.1. 2019. URL: https://www.npmjs.com/package/glob (besucht am 22. 10. 2019) (siehe S. 83).

[119] Yusuke Suzuki. *Escodegen - ECMAScript code generator*. Version 1.12. 2015. URL: https://github.com/estools/escodegen (besucht am 11. 10. 2019) (siehe S. 24).

[130] Nicholas C. Zakas. *ESLint - Pluggable JavaScript linter*. Version 6.5. 2019. URL: https://eslint.org (besucht am 06. 10. 2019) (siehe S. 36).

[133] Henry Zhu und Oleg Gaidarenko. *JavaScript Concrete Syntax Tree*. Version 0.4. 2019. URL: https://github.com/cst/cst (besucht am 11. 11. 2019) (siehe S. 88).

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

A Anhang

A.1 Quelltext des umgesetzten Transpilers

Der vollständige Quelltext des im Zuge dieser Arbeit entwickelten Transpilers zur Übersetzung von Flow nach TypeScript ist unter folgender Adresse einsehbar. Das Projekt wurde unter der MIT-Lizenz [97] veröffentlicht:

https://github.com/grubersjoe/reflow

A.2 Quelltext der angepassten Version von Prettier

Auch der Quelltext der geringfügig angepassten Version von *Prettier*[43] [38] wurde auf GitHub veröffentlicht:

https://github.com/grubersjoe/prettier-reflow

[43] Vgl. Abschnitt 4.5.

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 18. Dezember 2019

_____

Jonathan Gruber