Debuggen von Quelltexttransformationen

Tom Braun
Hasso-Plattner Institut

ZUSAMMENFASSUNG

BabelJS ist ein JavaScript Quelltext zu Quelltext Compiler der zur Transformation neuerer JavaScript Syntax in ältere JavaScript Syntax oder benutzerdefinierter Transformationen dient. BabelJS ist ein wichtiges Werkzeug in der modernen Webentwicklung und findet dort in fast jedem Projekt Verwendung. Die Transformationen werden über Plugins gesteuert, die allerdings nicht leicht zu schreiben sind. Plugins können aus den verschiedensten Gründen nicht funktionieren und mithilfe herkömmlicher Debugging Werkzeuge ist es mitunter sehr schwer die Ursache der Probleme zu finden. Die Schwierigkeit entsteht dadurch, dass Fehlerursache und Wirkung weit voneinander entfernt liegen und sich die Fehlerursache daher kaum nachvollziehen lässt.

Um den Fehlerlokalisierungsprozess zu erleichtern wird in dieser Arbeit ein dediziertes Werkzeug und dessen Implementation beschrieben. Das Werkzeug zeichnet die Ausführung der Plugins auf und ermöglicht im Nachhinein die Aufzeichnung beliebig vorwärts und rückwärts zu explorieren. Dadurch lässt sich die Auswirkung eines Fehlers besser zu seiner Quelle zurückverfolgen. Wie das Werkzeug Entwickler unterstützen kann wird anhand eines Beispielproblems illustriert. Ich bin davon überzeugt, dass das Werkzeug den Fehlerfindungsprozess in Plugins beschleunigen und somit die Entwicklung von Plugins beschleunigen kann.

1 EINFÜHRUNG

Aus der modernen Javascriptentwicklung ist BabelJS [7] kaum mehr wegzudenken. Es ist ein weit genutztes Werkzeug, das dazu dient neuere Javascript Syntax in ältere Javascript Syntax umzuwandeln, damit möglichst viele Browser damit umgehen können. Auch ermöglicht es eigene Quelltextransformationen vorzunehmen. Dazu nutzt BabelJS eine Pluginarchitektur, das heißt man kann mithilfe von Plugins die Transformationen steuern.

BabelJS Plugins schreiben ist eine nicht triviale Aufgabe. Zuallererst unterliegen Plugins wie jeder Quelltext Fehlern. Auch bereiten Seiteneffekte zwischen Plugins oder bisher nicht bedachte Syntaxkombinationen Probleme, da diese mitunter erst entdeckt werden, wenn Transformationen auf großen Mengen Quelltext angewendet werden.

Werden große Mengen Quelltext transformiert kommen herkömmlichen Fehlerbehebungsmechanismen schnell an ihre Grenzen. Ein Grund dafür ist, dass nur das Endergebnis einer potentiell langen Kette an Transformationen ohne Zwischenschritte ausgegeben wird. Sollte in einem Zwischenschritt eine Fehler entstanden sein, können Entwickler mit Fehlermeldungen zu im ursprünglichen Quelltext nicht vorkommenden Teilen konfrontiert werden. Mitunter verläuft die Transformation auch ohne Fehler, produziert aber Quelltext mit Logik- oder Syntaxfehlern. Um Entwicklern die Entwicklung solcher Plugins zu erleichtern wäre bessere Werkzeugunterstützung wünschenswert.

Diese Arbeit behandelt das Konzept und eine konkrete Implementation für ein Werkzeug zur Fehlerfindung in BabelJS-Plugins. Dieses schneidet die komplette Ausführung von Plugins mit. Die Aufnahme umfasst die Instruktionen der Plugins sowie die Transformationen, die der auslösenden Instruktion zugeordnet werden. Eine anschließende Visualisierung ermöglicht es den kompletten Verlauf in beliebiger Reihenfolge zu betrachten. Wird während der Transformation ein Fehler geworfen kann man sich beispielsweise die Aufnahme bis zum Fehlerzeitpunkt ansehen. Von dort aus kann rückwärts betrachtet werden, welche Instruktionen ausgeführt wurden und welchen Einfluss sie auf die Transformation hatten. Dadurch können eventuell Erkenntnisse aus Zwischenergebnissen gezogen werden, die bei der Lokalisierung der Fehlerursache helfen.

Im Verlauf dieser Arbeit wird in Abschnitt 2 BabelJS, die Probleme der Pluginentwicklung und ein Beispiel, in dem versucht wird eine Fehlerursache mithilfe herkömmlicher Debugging Techniken zu lösen, vorgestellt. In Abschnitt 3 wird ein Lösungsansatz vorgestellt, dessen konkrete Implementation in Abschnitt 4 thematisiert wird. Um die Nützlichkeit des Werkzeuges zu zeigen, wird das Beispielproblem aus Abschnitt 2 in Abschnitt 5 mithilfe der Implementation gelöst. Anschließend werde verwandte Arbeiten in Abschnitt 6 diskutiert. Schließlich präsentiert Abschnitt 7 Limitierungen und Ausblick des aktuellen Ansatzes und Abschnitt 8 gibt ein Fazit.

2 HINTERGRUND

Um die Probleme beim Entwickeln von BabelJS Plugins besser verstehen zu können wird zunächst eine grobe Übersicht über BabelJSs Funktionsweise gegeben. Anschließend werden konkrete Probleme bei der Entwicklung von Plugins beleuchtet und anhand eines Beispiels die Probleme herkömmlicher Fehlerfindungswerkzeuge gezeigt.

2.1 BabelJS

BabelJS ist ein in JavasScript geschriebener Quelltext zu Quelltext Compiler, der häufig zum Umwandeln von ECMAScript 2015+ Quelltext in älteren ECMAScript Quelltext verwendet wird. Weiterhin ist es möglich eigene Quelltexttransformationen zu definieren. Dafür bedient BabelJS sich einer Pluginarchitektur. Entwickler können Plugins entwickeln, die die Transformation steuern.

Quelltexttransformationen erfolgen nach dem folgenden Schema [4], das auch in Abbildung 1 gezeigt wird:

(1) Parsen: In dieser Phase wird Quelltext als Eingabe entgegengenommen und in einen abstrakten Syntaxbaum umgewandelt.
(2) Transformieren: Der abstrakte Syntaxbaum wird Knoten für Knoten traversiert. Plugins werden auf die besuchten Knoten angewandt
(3) Generieren: Der transformierten abstrakten Syntaxbaum wird wieder zu Javascript Quelltext umgewandelt.

Für den Schritt des Parsens gilt es zu beachten, dass nur valide JavaScript Syntax oder einige vordefinierte Erweiterungen akzeptiert werden. BabelJS ermöglicht nicht syntaktisch neue Sprachen zu definieren.

Die Transformation wird wie bereits erwähnt durch Plugins gesteuert. Diese beinhalten jeweils eine Liste an Besuchermethoden [2]. Der Name der Methode stimmt mit der Typbezeichnung des Knotens des abstrakten Syntaxbaums überein. Über Seiteneffekte können in den Besuchermethoden Änderungen am abstrakten Syntaxbaum vorgenommen werden.

Eine vereinfachte Darstellung einer Besuchermethode die beispielsweise Bedingungen betrifft ist:

```
1 Conditional(node) {
2   const log = astFromString('console.log
      ("afterTest")')
3   node.test.insertAfter(log)
4 }
```

Die Menge an anzuwendenen Plugins wird BabelJS zu Beginn des gesamten Compilevorgangs als Liste übergeben wird. Plugins werden nicht seperat der Reihe nach angewendet. Stattdessen werden beim Besuchen eines Knotens die zutreffenden Methoden, sofern vorhanden, der Reihenfolge der Plugins in der Liste nach aufgerufen.

2.2 Probleme beim Schreiben von Babel Plugins

Das Herzstück von BabelJS sind die Plugins. Diese zu schreiben ist allerdings nicht leicht. Eine erste Hürde ist, dass sich

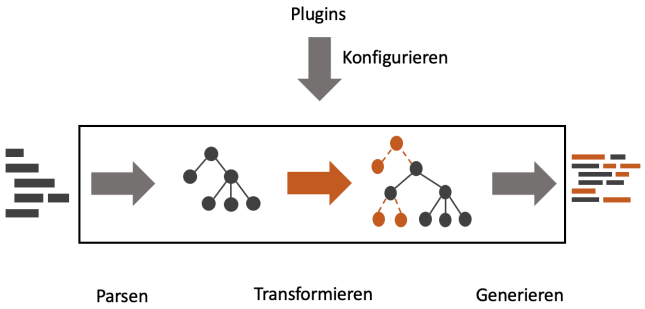

Abbildung 1: Überblick über den gesamten Compilevorgang von BabelJS.

bewusst gemacht werden muss, welche Knoten des abstrakten Syntaxbaum man wie verändern möchte. Häufig werden Kombinationen vergessen, da von über 100 Knoten [8] viele beliebig kombiniert werden können. Teilweise werden diese Kombinationen übersehen und erst in der Praxis gefunden.

Ein weiteres Problem ist, dass Plugins, die in Isolation fehlerfrei funktionieren in Kombination mit anderen Plugins Fehler verursachen können. Alle Plugins arbeiten auf einer gemeinsamen Datenstruktur, dem abstrakten Syntaxbaum, den sie durch Seiteneffekte verändern. Auch wenn sich die Entwickler von Plugins Mühe geben, kann es dennoch dazu kommen, dass Plugins die von anderen Plugins getroffenen Annahmen invalidieren. Die Fehlerquelle herauszufinden wird auch dadurch erschwert, dass Plugins nicht strikt nacheinander ausgeführt werden. Dies erschwert es Entwicklern über die Wirkung ihrer Plugins für sich nachzudenken. Sie können nicht Isolation betrachtet werden, sondern müssen im Zusammenspiel mit allen anderen Plugins durchdacht werden.

Fehler werden nicht immer während der Entwicklung entdeckt, sondern mitunter erst im Einsatz auf großen Mengen Quelltext. Da Fehler nicht immer in der Methode verursacht werden, die sie melden, hat man potenziell viele Besuchermethoden, die einen Fehler verursachen können. Tritt dieser Fall ein ist auch nicht immer eindeutig welche Besuchermethode die Ursache des Fehlers ist und Entwickler müssen diese unter großem Aufwand suchen. Es gilt zu bedenken, dass Entwickler nicht mit allen Plugins vertraut sind, was die Suche noch erschwert. Auch müssen Fehler nicht immer in einem offensichtlichen BabelJS Fehler resultieren. Es kann auch eine falsche Transformation vorgenommen worden sein. Dann kann es dazu kommen, dass fehlerhaftes Verhalten nicht in der Anwendungslogik, sondern an der Transformation liegt. In diesem Fall muss zunächst identifiziert werden, dass der Fehler durch Quelltexttransformationen verursacht wurde und anschließend wo der Fehler auftritt.

Fehlerquellen zu lokalisieren wird dadurch erschwert, dass BabelJS nur ein Endergebnis liefert. Entweder wird der Quelltext tranformiert oder es wird ein Fehler geworfen. Entwickler erhalten keinen direkten Zugang zu Zwischenständen. Diese können maximal im Debugger eingesehen werden. Dort werden aber nur die JavaScript Objekte angezeigt, durch welche navigiert werden muss und die interpretiert werden wollen. Es kann nicht einfach der aktuelle transformierte Quelltext eingesehen werden.

2.3 Debugging Babel Plugins

Um die beschriebenen Probleme greifbarer zu machen werden sie in diesem Abschnitt anhand eines Beispiels verdeutlicht. Der Quelltext der Beispielplugins wurde zum besseren Verständnis vereinfacht. Die Funktionalität der Methoden mag nicht sonderlich relevant erscheinen, ist allerdings eine starke Vereinfachung von Mechanismen für die Implementation eines in Abschnitt 4.2 beschriebenen Plugins.

Als Beispiel soll der folgende Quelltext transformiert werden:

```
1 let foo;
2
3 foo = 5;
4
5 if(true) { }
```

Angenommen es gäbe zwei BabelJS Plugins. Beide beinhalten jeweils nur eine Methode. Die erste ist:

```
1 AssignmentExpression(node) {
2  console.log(node.position.line)
3 }
```

Es wird die Zeile des aktuell besuchten Zuweisungsknotens auf die Konsole zur Ausführungszeit der Methode geschrieben. Bezogen auf den Quelltext oben würde einmal 3 ausgegeben werden, da sich die Zuweisung *foo = 5* in Zeile 3 befindet.

Die Methode des zweiten Plugins ist:

```
1 Conditional(node) {
2  const log = astFromString('console.log
     ("afterTest")')
3  node.test.insertAfter(log)
4 }
```

Es wird nach dem Test einer Bedingung die Anweisung *console.log("afterTest")* eingefügt. Wird der transformierte Quelltext ausgeführt, wird wie erwartet afterTest auf die Konsole geschrieben.

Kombiniert werfen die beiden Plugins einen Fehler. Dieser besagt für das erste Plugin sei die Position des Knotens nicht bekannt und es könne dem entsprechend nicht auf deren Zeile zugegriffen werden. Ein erster Ansatz die Ursache zu finden, wäre es einen Haltepunkt in die Methode *AssignmentExpression* zu setzen. Dies ergibt, dass die Methode ein erstes Mal erfolgreich durchlaufen und dann der Haltepunkt unerwartet noch ein zweites mal ausgelöst wird. Diesmal kann im Debugger betrachtet werden, dass der besuchte Knoten keine Position besitzt und als Folge daraus der Fehler geworfen wird. Es bleibt allerdings die Frage, woher die Zuweisung kommt.

In diesem minimalen Beispiel kann als Fehler der zweite Besucher deduziert werden. Dies wäre in der Praxis mit mehreren Besuchermethoden mit potenziell mehr als zwei Plugins kaum möglich. Dort könnte es eine beliebige Kombination sein, was den Aufwand stark in die Höhe treibt und schließlich ohne weitere Hinweise unpraktikabel macht.

Als nächster Versuch kann in die zweite Methode ein Haltepunkt gesetzt und die Ausführung der Plugins erneut angestoßen werden. Der erste Haltepunkt für die Zuweisung *foo = 5* wird ausgelöst. Als nächstes wird der Haltepunkt für *if(true* ausgelöst. Da dieser vor dem zweiten Auslösen des Haltepunktes in *AssignmentExpression* liegt, muss der zweite Knoten hier erstellt werden. In Zeile 2 wird nur ein Knoten erstellt und danach sieht der abstrakten Syntaxbaum immer noch genau so aus wie davor. Der Fehler muss also in Zeile 3 entstehen.

Es wurde herausgefunden welche Zeile den neuen Knoten in den abstrakten Syntaxbaum einfügt, aber noch nicht warum. Die von BabelJS genutzte Methode, die in Zeile 3 Seiteneffekte auslöst, ist *insertAfter*. Nach dieser Instruktion lässt sich eine Änderung am abstrakten Syntaxbaum beobachten. Der Inhalt des Attributes *test* des aktuellen Knotens wurde nicht nur um *console.log("afterTest")* erweitert, sondern auch um eine Zuweisung. Diese Beobachtung erfordert aber genaues Betrachten des abstrakten Syntaxbaums im Debugger. Ein Vorgang der mit viel Interpretation verbunden ist, da nur eine verschachtelte Objektstruktur angezeigt wird, wie in Abbildung 2 zu sehen ist.

Der Fehler ließ sich in diesem einfachen Beispiel mit etwas Mühe ausfindig machen. In der Praxis wäre dieser Ansatz aufgrund von mehr Besuchermethoden sowie eines potenziell längeren Eingabequelltextes, der in einem komplexeren abstrakten Syntaxbaum resultierten würde, nicht praktikabel.

* * *

Anhand des Beispiels sollte klar geworden sein, dass weitere Werkzeugunterstützung wünschenswert ist. Es wurde gezeigt, dass herkömmliches Debugging mithilfe von Haltepunkten nicht ausreichend ist. Aus Fehlermeldungen kann nur abgeleitet werden welche konkrete Besuchermethode einen Fehler wirft, nicht aber welche Besuchermethoden vorher aufgerufen wurden. Wie der Fehler entstand wird

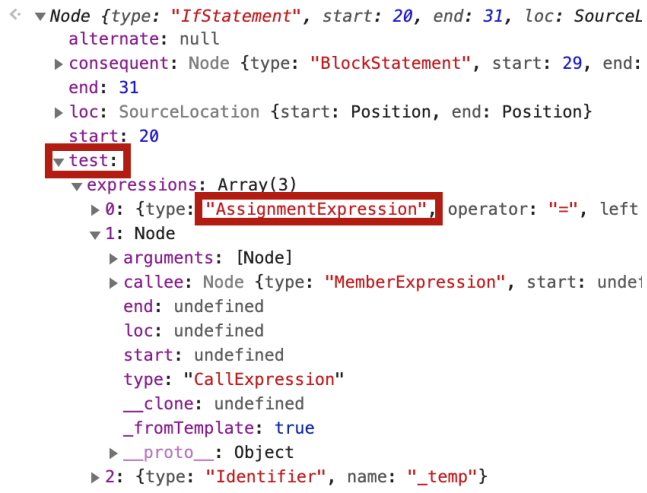

Abbildung 2: Der Knoten des abstrakten Syntaxbaums der Bedingung nachdem *insertAfter* aus dem zweiten Plugin ausgeführt wurde. Markiert wurde der *test*-teil des Knotens der aus mehreren Ausdrücken besteht wovon der erste eine Zuweisung ist. Der Ausschnitt wurde bereits gekürzt.

nicht unbedingt aufgedeckt, da keine Zwischenergebnisse ausgegeben werden und von Haltepunkten in JavaScript nur spätere Instruktionen beobachtet werden können. Außerdem würde es die schiere Menge an Knoten, die ein realistisches Quelltextbeispiel erzeugen würde, zu einer unpraktikablen Aufgabe machen jeden Knoten anhand dieses Ansatzes zu inspizieren. Der gesamte Vorgang wird durch die Unzugänglichkeit der Darstellung des abstrakten Syntaxbaum erschwert.

3 ANSATZ

Aufgrund der aufgezeigten Probleme wird in dieser Arbeit ein Werkzeug vorgeschlagen, das diese angeht. Das vorgeschlagene Werkzeug ist kein allgemeiner Debugger sondern ein für BabelJS dediziertes Werkzeug. Es soll in der Lage sein die gesamte Ausführung der Plugins explorierbar zu machen. Gleichzeitig sollte sich auch nur auf Plugins konzentriert werden, da BabelJS interne Funktionen außerhalb der Zuständigkeit der Pluginentwickler liegen und somit für sie unrelevant ist.

Die Idee ist eine Art Omniscient Debugging [5] zu verwenden. Omniscient Debugging beschreibt ein Werkzeug, das jede Zustandsänderung und jeden Methodenaufruf aufzeichnet. Diese Aufzeichnung kann danach in beliebiger Reihenfolge immer wieder vorwärts und rückwärts abgespielt werden. Dies soll in den Kontext von BabelJS übertragen werden. Um die Ausführung des Plugins aufzuzeichnen müssen ausgeführte Instruktionen der BabelJS Plugins zur Laufzeit aufgezeichnet werden. Indem nur die Instruktionen der Plugins aufgezeichnet werden, können sich Pluginentwickler auf ihre Domäne konzentrieren.

Außerdem werden die Veränderungen des abstrakten Syntaxbaums aufgezeichnet. Diese werden durch die Instruktionen der Plugins ausgelöst. Durch das Mitschneiden der Veränderungen und Einordnung in die Aufzeichnung der Instruktionen kann zugeordnet werden welche Instruktion, welche Veränderung ausgelöst hat. Die Aufzeichnungen der Veränderungen können später verwendet werden um Zwischenschritte der Transformation explorierbar zu machen.

Liegen die gesammelten Aufzeichnungen der Instruktionen und Veränderungen vor müssen diese fehlersuchenden Entwicklern zugängig gemacht werden. Eine einfache textuelle Ausgabe wäre möglich, aber vermutlich aufgrund der anfallenden Datenmenge und deren Natur sehr unübersichtlich. Die Daten unterscheiden sich vom mentalen Modell der Programmierer und eine besserer Darstellung würde sie unterstützen. Beispielsweise möchte man einen Knoten des abstrakten Syntaxbaums manchmal als Programmierobjekt und manchmal als dazugehöriger Quelltext betrachten. Aber auch als Objekt interessieren manche Informationen nie, da diese nur Implementationsdetail sind. Es ist also eine flexiblere Form der Darstellung gefragt. Abbildung 3 zeigt wie ein Mockup für so eine Darstellung aussehen könnte. Der Vorschlag ist eine graphische Benutzeroberfläche mit mindestens einer Möglichkeit zwischen den Instruktionen zu navigieren und den dazugehörigen Pluginquelltext angezeigt zu bekommen. Weiterhin sollten die dazugehörigen Veränderungen des abstrakten Syntaxbaums visualisiert werden. Sei es in Form einer hierarchischen Ansicht oder direkt der Quelltext inklusive der Veränderungen bis zu diesem Punkt. Dadurch kann feingranular eingesehen werden, welches Plugin mit welcher Instruktion welche Veränderung ausgelöst hat. Zusätzlich liegt die gesamte Historie vor. Soll herausgefunden werden an welchem Zeitpunkt Knoten eingefügt wurde kann die Aufzeichnung rückwärts durchsucht werden. Es ist nicht mehr die Frage ob die gesuchte Stelle nach dem aktuellen Haltepunkt kommt, sondern wo in der Aufzeichnung sie ist. Eine Frage, die gegebenenfalls durch geeignete Suchelemente des Werkzeuges vereinfacht werden kann. Wie diese aussehen liegt allerdings außerhalb des Umfangs dieser Arbeit.

4 IMPLEMENTATION

Es liegt eine Beispielimplementation des vorgschlagenen Ansatzes vor. Dieser wurde in Lively 4 [6] implementiert. Lively ist eine auf JavaScript basierenden Liveprogrammierumgebung, die im Browser läuft. In diesem Abschnitt wird das grobe Vorgehen der Implementation sowie einige wichtige Details genauer erläutert.

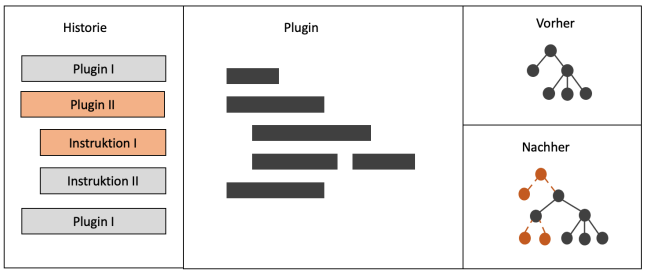

Abbildung 3: Mockup der Visualisierung. Orange markierte Instruktionen lösen eine Veränderung am abstrakten Syntaxbaum aus. Die Veränderung wird in der rechten Spalte einmal in der Version vor der Instruktion und nach der Instruktion gezeigt. Änderungen werden auch hier orange markiert.

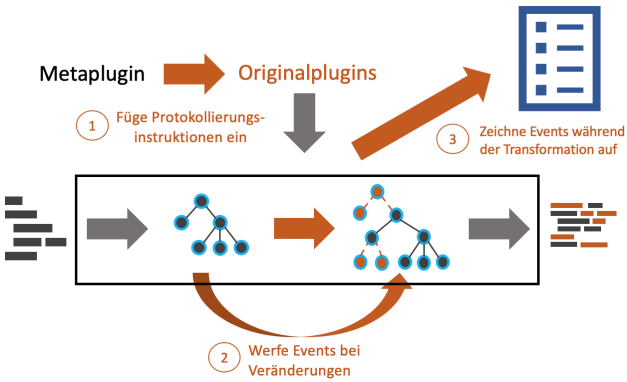

Abbildung 4: Überblick über den modifizierten Compilevorgang.

4.1 Vorgehen

Die Eingabe für die Analyse ist Quelltext und eine Liste an Plugins. Zunächst werden die Plugins (Originalplugins) mithilfe eines speziellen Plugins (Metaplugin), das in Abschnitt 4.2 genauer beschrieben wird, transformiert. Dies fügt Protokollierungsinstruktionen in die Originalplugins ein. Die Instruktionen ermöglichen es später die ausgeführten Instruktionen der Originalplugins nachzuvollziehen.

Als nächstes wird der gegebene Quelltext in einen abstrakten Syntaxbaum umgewandelt. Währenddessen werden allen Knoten des abstrakten Syntaxbaum eindeutige Kennnummern zugeordnet. Außerdem werden alle Knoten mit Adaptern versehen, die jeglicher Veränderungen melden. Wie die Adapter funktionieren wird in Abschnitt 4.3 erläutert.

Schließlich wird der modifizierte abstrakten Syntaxbaum mit den transformierten Plugins transformiert. Während dieser Transformation erfassen die Protokollierungsinstruktionen welche Instruktionen der Originalplugins ausgeführt wurden und die Adapter verzeichnen alle am abstrakten Syntaxbaum auftretenden Änderungen.

Schließlich werden die aufgezeichneten Informationen vorprozessiert und eine Visualisierung erstellt. Die Visualisierung wird in Abschnitt 4.4 genauer vorgestellt.

4.2 Aufzeichnen der Instruktionen

Es sollen ausgeführte Instruktionen aufgezeichnet werden. Dazu gehören sowohl Statements als auch Funktionsaufrufe. Da die Implementation im Rahmen von Lively 4 unmodifiziert im Browser laufen soll, ist es nicht möglich auf Bytecodeebene oder VM-ebene die Instruktionen zu verfolgen. Es bleibt daher ein Aufzeichnen direkt im Quelltext. Eine Anforderung ist, dass auch bestehende Plugins ohne weiteren Aufwand analysiert werden können. Sie im Nachhinein manuell zu verändern wäre zu viel Aufwand um praktikabel zu sein. Auch ist es nicht ausreichend die Babelfunktionen, wie beispielsweise *insertAfter*, für Aufzeichnungen zu verändern. Kontrollflussinstruktionen wären davon nicht betroffen, obwohl Informationen, wie ein vorzeitiger Abbruch einer Besuchermethode, relevant sind.

Der gewählte Ansatz nutzt BabelJS. Es ist dazu in der Lage Quelltext dahingehend zu verändern, dass dieser Informationen über sich zur Laufzeit abgibt. Dafür ist die Entwicklung eines Plugins nötig, welches vor und nach Kontrollflussinstruktionen sowie Funktionsaufrufen Instruktionen einfügt. Diese enthalten spezifische Informationen über die Art der aufgenommenen Instruktion, bei Bedarf eventuelle Parameter oder Argumente und zu welcher Position im originalen der aktuelle Knoten korrespondiert.

Das Prinzip soll am folgenden Beispiel verdeutlicht werden. Der Ausgangsquelltext des Plugins ist:

```
1 AssignmentExpression(node) {
2  console.log(node.position)
3 }
```

In diesem Beispiel würden folgende Informationen über die Ausführung gesammelt: die Methode *AssignmentExpression* wird betreten, die Methode log wird aufgerufen, die Methode log wurde verlassen, die Methode *AssignmentExpression* wird verlassen. Außerdem wird für jede Instruktion die Position im originalen Quelltext gespeichert. Die Position ist in Form einer Zahl kodiert und der erste Parameter der eingefügten Methoden. Der resultierende transformierte Quelltext würde vereinfacht wie folgt aussehen:

```
1 AssignmentExpression(node) {
2  trace.enterFunction(0)
3
4  trace.aboutToEnterFunction(1,'log')
5  console.log(node.position)
6  trace.leftFunction(1,'log')
7
```

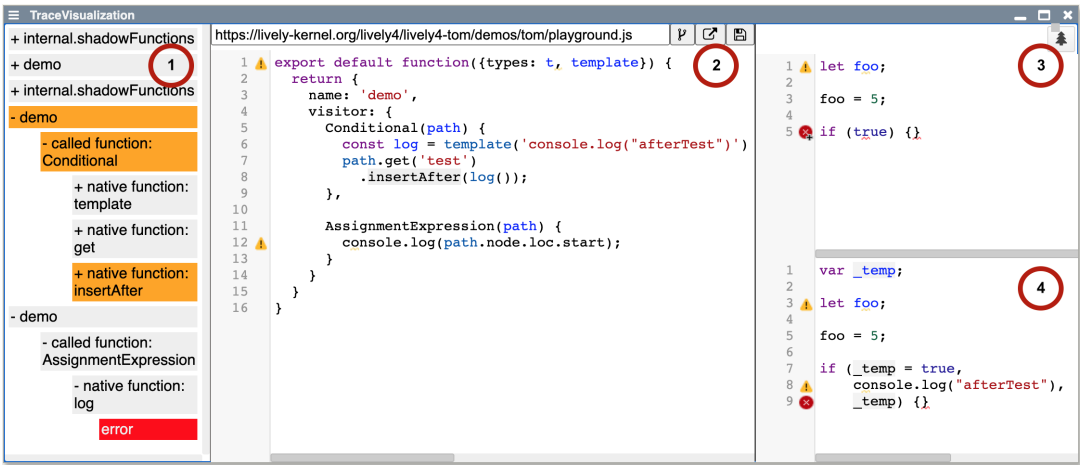

Abbildung 5: Implementation des vorgeschlagenen Werkzeuges. Angezeigt wird der Zustand für die Instruktion *insertAfter*.

```
  trace.leaveFunction(0)
}
```

4.3 Mitschneiden der Veränderungen des abstrakten Syntaxbaums

Der abstrakte Syntaxbaum liegt in Form eines Objektgraphen vor, der nur durch die Seiteneffekte der Plugins verändert wird. Die Aufgabe ist also jegliche Veränderungen an diesen Objekten festzustellen. Der naive Ansatz wäre es nach jeder Veränderung den kompletten abstrakten Syntaxbaum zu kopieren. In der Praxis führt dies zu sehr langen Ausführungszeiten bis hin zum Aufhängen der ausführenden Website. Es muss daher eine effizienterer Ansatz gefunden werden.

Wie bereits vorher beschrieben wird jeder Knoten des abstrakten Syntaxbaum mit einem Adapter versehen. Dies erfolgt vor der Transformation. Die Attribute der Knoten werden durch JavaScript *getter* [1] und *setter* [1] ersetzt. Diese ermöglichen es den Zugriff auf Attribute zu Verändern, ohne dass das syntaktischen Einfluss auf den restlichen Quelltext, und somit wie BabelJS intern die Knoten verändert, hat. Es ist möglich in *setter*n den neuen Wert abzufangen und aufzuzeichnen. Die *getter* werden nur von JavaScript benötigt um weiterhin auf die Attribute zugreifen zu können. Sie beinhalten kein verändertes Verhalten. Arrays müssen speziell behandelt werden, da es nicht möglich ist auf ihnen *getter* und *setter* in gewünschter Art und Weise zu implementieren. Daher werden sie durch JavaScript Proxies [1] ersetzt. Sie ermöglichen es alle Zugriffe auf ein gebundenes Objekt abzufangen. Alle Veränderungen an dem Array können somit mitgeschrieben werden.

Mithilfe der Adapter ist es möglich Veränderungen zu erfassen. Diese müssen allerdings kopiert werden. Sollte das Kopieren vernachlässigt werden, könnten spätere Seiteneffekte die mitgeschriebenen Werte für diesen Zeitpunkt verfälschen. Werte wie Nummern und Zeichenfolgen können effizient kopiert werden. Bei Objekten, die keine Knoten des abstrakten Syntaxbaum sind wird angenommen sie seien nicht besonders groß und es wird eine vollständige Kopie angefertigt. Knoten des abstrakten Syntaxbaum stellen eine Herausforderung dar. Enthalten sie große Teilbäume ist es möglich in ein ähnliches Effizienproblem wie beim Kopieren des gesamten abstrakten Syntaxbaum zu laufen. Daher werden nur Knoten ohne Kennnummer kopiert. Sie waren nicht im ursprünglichen abstrakten Syntaxbaum enthalten und sind somit neu. Neue Knoten und neue Knoten in deren Teilbäumen werden kopiert. Wird auf einen nicht Knoten gestoßen wird damit wie oben beschrieben verfahren. Wird auf einen bekannten Knoten gestoßen wird dessen Kennnummer, in einem speziellen Objekt gespeichert. Dieses Objekt kann später beim Wiederherstellen von Änderungen erkannt und anhand der Kennnummer die Referenz auf den bereits bekannten Knoten aufgelöst werden. Schließlich werden die Knoten mit Adaptern und einer Kennnummer versehen. Dadurch können auch an ihnen Änderungen mitgeschrieben werden.

4.4 Visualisierung

Die Visualisierung dient der Explorierbarkeit der aufgezeichneten Informationen. Es müssen sowohl die Instruktionen mit eventuell daran gebundenen Änderungen des abstrakten Syntaxbaum vorhanden sein, als auch wo diese sich in den Plugins befinden und wie der abstrakten Syntaxbaum danach aussieht, beziehungsweise wie der transformierte Quelltext danach aussieht.

In der Abbildung 5 wird die aktuelle Implementation gezeigt. Die Visualisierung ist in einem dreispaltigen Layout

angelegt. In der ersten von links (1) ist die Historie der Instruktionen. Diese sind verschachtelt. Oberknoten für Verschachtelungen sind beispielsweise Plugins oder Funktionen. Weitere Instruktionen aber auch wieder mögliche Verschachtelungen lassen sich darunter finden. Instruktionen, die Veränderungen am abstrakten Syntaxbaum vornehmen, und alle Knoten in der Verschachtelung darüber sind für bessere Navigierbarkeit zu solchen Knoten orange eingefärbt. Fehler werden rot eingefärbt.

Schwebt die Maus über einer Instruktion wird in der mittleren Spalte (2) der Pluginquelltext, aus dem die Instruktion stammt, angezeigt und die konkrete Instruktion farblich markiert. Dadurch kann der Kontrollfluss Instruktion für Instruktion nachvollzogen werden.

Das Schweben der Maus über einer Instruktion verändert potenziell auch die rechte Spalte. Diese zeigt oben (3) den Quelltext, wie er vor der aktuellen Instruktion und unten (4) danach aussieht. Dadurch kann der Einfluss von Instruktionen direkt betrachtet und Zwischentransformationen einsehbar gemacht werden. Direkt darüber existiert auch ein Button der ein Fenster mit dem jeweils zu dem vorherigen oder neuen Quelltext korrespondierenden abstrakten Syntaxbaum öffnet.

5 EVALUATION

Offen bleibt die Frage inwiefern das implementierte Werkzeug zur Fehlerlokalisierung geeignet ist. Zur Evaluation soll das auf Abschnitt 2.3 bekannte Beispiel erneut aufgegriffen werden.

Wird das in dieser Arbeit vorgeschlagene Werkzeug ausgeführt, kann der transformierte Quelltext direkt vor dem Fehler inspiziert werden. Dieser sieht wie folgt aus:

```
1 var _temp;
2 let foo;
3
4 foo = 5;
5
6 if(_temp = true, console.log("afterTest"
      ), _temp) { }
```

Es wird direkt klar, dass der Fehler von der neuen Zuweisung in der Bedingung kommen muss. Um nachzuvollziehen woher diese kommt, kann die Historie durchsucht werden. Nach Aufklappen der Historie ist sichtbar, dass die Instruktion *insertAfter* aus dem zweiten Plugin die einzige mit Veränderungen am abstrakten Syntaxbaum ist. Hier wird die zweite Zuweisung ein den abstrakten Syntaxbaum eingefügt. Zu dieser Erkenntnis gelangt man diesmal allerdings ohne mehrmaliges neu Starten des Debuggers und ohne den abstrakten Syntaxbaum aus Objekten interpretieren zu müssen. Es kann die gesamte Ausführung vorwärts und rückwärts betrachtet werden. Es ist exakt aufgelöst welche Instruktion welche Veränderung am abstrakten Syntaxbaum auslöst und wie er an einer beliebigen Instruktion aussieht. Problematisch ist nur, dass große Plugins oder viel Quelltext eine große Menge an Ausgabe erzeugen, die Durchgesehen werden muss. Eine Hilfe dafür ist die farbliche Markierung von Abschnitten mit Änderungen, diese können aber in der Praxis immer noch zu groß sein.

6 VERWANDTE ARBEITEN

Der vorgeschlagene Ansatz befasst sich damit die Fehlerfindung in BabelJS Plugins zu vereinfachen. Damit fällt der Ansatz in eine Nische, es gibt aber auch allgemeinere Ansätze deren Unterschiede zu dem vorgeschlagenen in diesem Kapitel erläutert werden sollen. An dieser Stelle sollen Omniscient Debugging [5] und Whyline [3] genannt werden.

Der vorgeschlagene Ansatz orientiert sich am Omniscient Debugging. Wie beim Omniscient Debugging wird der Instruktionsverlauf aufgezeichnet. Anders als beim Omniscient Debugging aber nicht der Zustand. Stattdessen bedient sich der vorgeschlagene Ansatz Domänenwissen und zeichnet die Veränderungen des abstrakten Syntaxbaum auf. Die Hoffnung dahinter ist mögliche Performanzprobleme des Omniscient Debuggings zu umgehen. Gleichzeitig schränkt die Nutzung des Domänenwissens auch den Suchraum ein, da dadurch unrelevante Instruktionen, wie BabelJS interne Funktionen, nicht mit aufgezeichnet werden. Dies steht im Gegensatz zum Omniscient Debugging, das einen mit dem kompletten Zustand konfrontiert.

Die Whyline ermöglicht es Fragen über die Ausführung zu stellen. Im Vordergrund stehen die beiden Fragen: Warum ist die Änderung aufgetreten? Warum ist die Änderung nicht aufgetreten? Die Whyline ermöglicht es diese Fragen direkt zu beliebigen Änderungen zu stellen. Die Fragen werden dann mithilfe einer Kombination aus statischer und dynamischer Analyse beantwortet. Schließlich kann die Frage durch die in der Analyse gewonnenen Informationen beantwortet werden. Das in dieser Arbeit vorgeschlagene Werkzeug nutzt im Vergleich nur dynamische Analyse und kann beantworten warum eine Änderung am abstrakten Syntaxbaum vorgenommen wurde, wenngleich die Antwort in der Aufnahme gesucht werden muss. Das vorgeschlagene Werkzeug ist nicht in der Lage zu beantworten warum eine Änderung nicht vorgenommen wurde.

7 LIMITIERUNGEN UND AUSBLICK

Aktuell hat das in dieser Arbeit vorgeschlagene Werkzeug noch Defizite gegenüber einem allgemeinen Debugger und bringt seine eigenen Nachteile mit. Ein normaler Debugger ist weiterhin in der Lage Zustand anzuzeigen und diesen Verändern zu können. Das Werkzeug erstellt mitunter sehr

große Aufzeichnungen, die unübersichtlich sind. In diesem Abschnitt werden Vorschläge gemacht, wie diese Punkte angegangen werden könnten.

Anfragen Große Aufzeichnungen müssen besser explorierbar gemacht werden. Die farbliche Markierung von Instruktionen, die Änderungen am abstrakten Syntaxbaum auslösen sind ein Schritt in diese Richtung. Sie ermöglichen eine bessere Selektierung nach potenziell wichtigen Informationen, sind für große Aufzeichnungen aber nicht ausreichend. Ein Nutzer muss sich potenziell durch viele Ebenen klicken und sehr viele Informationen sichten. Besser wäre es Anfragen auf der Aufzeichnung stellen zu können. Nach den Anfragen wird nur die Antwort gezeigt. Dadurch könnten schnell interessante Informationen herausgefiltert und unwichtige versteckt werden. Vorstellbar wären beispielsweise folgende Anfragen: Welche Methoden verändern den abstrakten Syntaxbaum? Welche Methoden brechen vorzeitig ohne Veränderungen ab? Welche Instruktionen verändern einen konkreten Knoten?

Anzeigen von Zustand

Aktuell kann eingesehen werden welche Instruktionen ausgeführt wurden, mitunter ist für Programmierende aber auch interessant wieso Instruktionen ausgeführt wurden. Dafür ist es mitunter nötig den Zustand von Variablen einsehen zu können. Dieser wird aktuell nicht erfasst, da der Zustand kopiert werden müsste und das zu einem potenziellen Performanzproblem führen kann. Würden beispielsweise ungefiltert die Argumente von Methoden gespeichert, würden für die Besuchermethoden der aktuelle Knoten inklusive des Teilbaumes darunter kopiert. Dies ist exakt das Problem welches in der Implementation mit den Adaptern umgangen wird und hier neu eingeführt werden würde.

Eine einfache Variante die Kopien zu implementieren wäre es nicht alle Strukturen zuzulassen und selektierte Objekte, wie Knoten des abstrakten Syntaxbaums, durch Platzhalter zu ersetzen. Würde sich in der Praxis herausstellen die Informationen werden doch benötigt oder das Kopieren ist zu aufwendig, könnten die Informationen auch selektiv gewonnen werden. Im Hintergrund wird die Transformation neu gestartet, unter der Annahme das die Transformationen immer einigermaßen schnell vonstatten gehen, und nur die benötige Kopie erstellt. Dieser Prozess würde immer auf Anfrage des Nutzers angestoßen. Da nur verhältnismäßig wenige Daten kopiert werden sollte dies nicht allzu lange dauern.

Verändern von Zustand

Normale Debugger sind in der Lage den Zustand zu einem beliebigen Zeitpunkt der Ausführung zu verändern. Dies wäre auch für das vorgeschlagene Werkzeug wünschenswert. Dadurch könnte direkt ausprobiert werden, wie ein anderer Zustand die Transformation verändert hätte und folglich ob diese Änderung generell sinnvoll wäre. Dieser Vorschlag geht einher mit der vorherigen Implementation den Zustand explorierbar zu machen.

8 FAZIT

Das Schreiben von BabelJS Plugins in ein nicht trivialer Vorgang, der mit herkömmlichen Debugging Techniken nur unzureichend unterstützt werden kann. In dieser Arbeit wurde ein Ansatz zur besseren Fehlerlokalisierung ausgearbeitet. Der Ansatz ermöglicht wie beim Omniscient Debugging die Instruktionen beliebig vorwärts und rückwärts zu betrachten. Dadurch kann vom Fehler aus besser nach der Ursache gesucht werden. Weiterhin wird der in der Arbeit vorgestellte Ansatz durch BabelJS spezifisches Wissen unterstützt. Dies schränkt die Nutzung des Werkzeuges auf BabelJS ein, ermöglicht aber in diesem Kontext genauere Informationen. In diesem Fall sind das Informationen darüber welche Instruktion welche Änderung am abstrakten Syntaxbaum vornimmt. Dadurch dass sich auf BabelJS spezifische Informationen beschränkt wird, werden Entwickler mit weniger und relevanteren Daten konfrontiert. Dennoch können Entwickler aktuell noch durch die Menge an Daten, die in einem realistischen Beispiel anfallen würden, überwältigt werden. Anfragen auf den Informationen wären daher der nächste Schritt, der in Betrachtung gezogen werden sollte.

LITERATUR

[1] ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification* (5.1 ed.). http://www.ecma-international.org/publications/standards/Ecma-262.htm

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional. ISBN: 0-201-63361-2 .

[3] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, Elizabeth Dykstra-Erickson and Manfred Tscheligi (Eds.). ACM, 151–158. https://doi.org/10.1145/985692.985712

[4] Jamie Kyle. [n.d.]. Babel Plugin Handbook. https://github.com/jamiebuilds/babel-handbook/blob/master/translations/en/plugin-handbook.md. Zugriff: 22.02.2021.

[5] Bil Lewis. 2003. Debugging Backwards in Time. *CoRR* cs.SE/0310016 (2003). http://arxiv.org/abs/cs/0310016

[6] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a live development experience for web-components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017*, Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara (Eds.). ACM, 28–35. https://dl.acm.org/citation.cfm?id=3167109

[7] Babel Team. [n.d.]. BabelJS. https://babeljs.io/. Zugriff: 21.02.2021.

[8] Babel Team. [n.d.]. BabelJS. https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md. Zugriff: 26.02.2021.