Typebasierte Analyse von JavaScript

Phillip Heidegger

Arbeitsbereich Programmiersprachen
Institut für Informatik
Universität Freiburg

1 Einleitung

Der AJAX Trend sorgt für eine starke Verbreitung von JavaScript und unterstreicht die Bedeutung der Sprache. Um die Zuverlässigkeit der entwickelten Software sicherzustellen und ein professioneller Einsatz von JavaScript zu ermöglichen, werden Analysewerkzeuge benötigt, wie diese für viele andere Programmiersprachen erhältlich sind.

Es gibt viele Ansätze, JavaScript sicherer zu machen. Eine Arbeit von Yu u.a. [7] schlägt z.B. eine dynamische Methode vor. Sie definiert eine Regelmenge, die JavaScript Quellcode dynamisch verändert. Dies ermöglicht dem Besucher einer Webseite, sich eine für ihn angepasste Regelmenge zu erstellen, die festlegt, welche Aktionen der JavaScript Quellcode ausführen darf. Bei einer solchen Lösung wird dem dynamischen Charakter von JavaScript Rechnung getragen und es werden Funktionen wie `eval`, `Document.write` u.a. unterstützt. Allerdings hilft eine solche Lösung einem Programmierer beim Finden von Sicherheitsproblemen seiner JavaScript–Programme nicht. Der Adressat des von Yu u.a. vorgestellten Ansatzes ist der Benutzer des Internetbrowsers.

Ist aber das Ziel, für einen JavaScript Programmierer Analysemethoden zu entwickeln, die ihm helfen seine Programme vor der Auslieferung an den Kunden auf Fehler zu analysieren, erfordert dies eine statische Analyse des Quellcodes. Als Beispiel sei hier ein Onlineportal einer Internetbank genannt. Dynamische Methoden würden erst nach Auslieferung oder bestenfalls beim Testen zu Ergebnissen führen. Bei einem Internetportal im sicherheitskritischen Umfeld ist dies nicht ausreichend, wie z.B. ein aktueller Artikel [5] über einen Onlineauftritt einer Internetbank zeigt. Hier gelingt es einem nach eigenen Aussagen nicht in Sicherheitstechnik ausgebildetem Reporter, in dem Internetauftritt einer Bank massive Sicherheitlücken zu finden.

Für statische Analysen muss angenommen werden, dass dem Programmierer der vollständige Quellcode seiner Anwendung statisch zur Verfügung steht. Dies bedeutet z.B., dass die Funktion `eval` mit statischen Analysen nicht behandelt werden kann.

Wie alle statischen Methoden, die nicht triviale Eigenschaften über Programmen berechnen, muss eine statische Analysen für JavaScript Näherungen durchführen. Hierbei gibt es zwei Möglichkeiten. Lindahl u. Sagonas [4] stellen in ihrem Artikel einen Ansatz mit Successtypen vor. Es werden nur Programme

zurückgewiesen, bei denen garantiert wird, dass an einer Stelle des Codes ein Fehler besteht. Falls nicht sicher ist, ob der Code korrekt ausgeführt werden kann, wird bei Successtypen das Programm nicht vom Typsystem zurückgewiesen.

Die zweite Möglichkeit besteht in einer konservativen Analyse des Quellcodes, der nur korrekte Programme bzgl. eines Typsystems als gültig einordnet. Für gültige Programme ist eine gewisse Menge Eigenschaften garantiert. So stellt das Typsystem von Java [2] z.B. sicher, dass bei einer Multiplikation keine Strings mit Zahlen multipliziert werden.

Hier wird die zweite Methodik gewählt und eine korrekte Analyse für die Scriptsprache JavaScript kurz vorgestellt. Würde die Eigenschaft, keine Zahlen mit Strings multiplizieren zu können, auf ein Typsystem für JavaScript übertragen, so würde der JavaScript Ausdruck `x = "2" * 4;` vom Typsystem als ungültig abgelehnt. Das Multiplizieren des Strings `"2"` und der Zahl 4 wäre unzulässig. In JavaScript werden Werte aber kontextabhängig in passende Werte anderen Typs konvertiert und der Ausdruck liefert die Zahl 8. Ein Typsystem, das die Multiplikation von Strings und Floats generell ablehnt, würde somit Programme zurückweisen, die ein korrektes Laufzeitverhalten besitzen. Dies wird sich im Allgemeinen nicht komplett vermeiden lassen, wenn man auf Korrektheit Wert legt, aber es muss zumindest sichergestellt werden, dass einfache Konvertierungen behandelt werden können. Der im Vortrag vorgestellte Ansatz basiert auf der Arbeit „Towards a Type System for Analysing JavaScript Programs“ von Peter Thiemann [6].

2 JavaScript

Vor der Vorstellung des Typsystems wird JavaScript anhand einiger Beispiele dargestellt. Hierdurch soll ein Gefühl für die Sprache vermittelt werden, und auf einige Fehlerquellen hingewiesen werden, die für einen Programmierer schwer zu erkennen sind.

Programm 1 JavaScript – automatische Konvertierung

```
1 var y = 0;
2 y.x = "Hallo";
3 alert(y.x);
```

Programm 1 stellt eine kurze Statementsequenz vor, in der ein schwer zu findender Fehler existiert. Das angegebene Programm erstellt eine Variable `y`, und weist dieser den Floatwert `0.0` zu. Durch die automatische Konvertierung wird in der nächsten Zeile einem Objekt, das durch Konvertierung der Zahl in ein Objekt entstanden ist, die Eigenschaft `x` gegeben, und diese auf den Wert `"Hallo"` gesetzt. Leider wird dieser Wert in der dritten Zeile nicht ausgegeben, denn das Objekt wurde nicht an die Variable `y` gebunden. Aus diesem Grund wird in Zeile drei nochmals ein zum Floatwert 0.0 passendes, anderes Objekt

erstellt. Dies besitzt keine Eigenschaft `x`. Somit erscheint `undefined` auf dem Bildschirm (Details siehe [6] oder [3]).

An dem Beispiel sieht man eine der zentralen Herausforderungen, denen sich ein Typsystem für JavaScript stellen muss. Objekte von JavaScript haben die Fähigkeit, dynamisch zur Laufzeit Ihre Eigenschaften, sowohl den Wert, als auch die Existenz der Eigenschaften, zu verändern. Sie besitzt, im Gegensatz zu den meisten objektorientierten Sprachen, keine Klassen, sondern es wird das Prinzip der Prototypen eingesetzt. Um Objekte zu erzeugen, kommen entweder Konstruktoren oder Objektliterale zum Einsatz.

Programm 2 JavaScript – Objekte – Konstruktor und Objektliteral

```
1 var proto = { test : "Eigenschaftswert" };
2 var f = function f(x) { this.x = x; return this; };
3 f.prototype = proto;
4 var o = new f("Test");
```

Wie Programm 2, Zeile 1 zeigt, ist das Objektliteral ist eine kompakte, aber nicht flexible Version, ein Objekt zu erstellen. Objektliterale ermöglichen es nicht, den Prototypen des erzeugten Objekts zu bestimmen. Es werden nur die angegebenen Eigenschaften durch den dem Doppelpunkt gefolgten Ausdruck initialisiert. Im Fall von `proto` wird die Eigenschaft ,,test“ auf den Stringwert ,,Eigenschaftswert“ gesetzt.

Bei der Objekterzeugung durch den Ausdrucks `new f("Test")` kommen die Prototypen zum Einsatz. Die Prototypreferenz des neu erzeugten Objekts wird durch den Konstruktor festgelegt, mit dessen Hilfe das Objekt erzeugt wird. In JavaScript sind Funktionen und Konstruktoren ebenfalls Objekte und können somit Eigenschaften besitzen. Wird ein Funktionsobjekt `f` mit dem Ausdruck `new f("Test")`, wie in Zeile 4, aufgerufen, erzeugt JavaScript ein neues Objekt im Speicher. Dieses Objekt erhält als Prototypreferenz den Wert der Eigenschaft ,,prototype“ des Objekts `f`. Somit zeigt die Prototypreferenz von `f` auf das Objekt `proto`. Die durch einen Konstruktor gesetzte Prototypreferenz und die Eigenschaft ,,prototype“ dürfen nicht verwechselt werden. Bei ,,prototype“ handelt es sich um eine Eigenschaft, die durch einen normalen Zugriff auf Eigenschaften gelesen und geschrieben werden kann. Die Prototypreferenz von `o` dagegen ist nicht für den Programmierer zugänglich und wird als implizite Prototypreferenz bezeichnet.[1]

Das Typsystem meiner Diplomarbeit [3] beinhaltet Union- und Intersectiontyps, behandelt die automatischen Konvertierungen von Werten in JavaScript, und kann mit den dynamischen Objekten und der Fehlerbehandlung vom Java-

[1] Der Standard [1] erlaubt dem Programmierer keinen Zugriff auf diese impliziten Prototype Pointer. Dies sieht bei den zur Zeit gebräuchlichen JavaScript Implementierungen anders aus, und kann einen möglicher weiteren Grund für Probleme bei einer Fehlersuch von JavaScript Code darstellen.

Script umgehen. Es stellt z.B. sicher, dass Fehler, wie im Programm 1 dargestellt, nicht möglich sind, aber es akzeptiert Ausdrücke wie `"2" * 4`. Es kann außerdem mit der Fehlerbehandlung von JavaScript umgehen und beherscht das Anspringen von gelabelten Statements mit `continue` und `break`.

3 Ausblick

3.1 Theorie

Soundness Für das logische Typesystem gibt es erste Schritte beim Preservation Beweis. Der noch fehlende Teil des Preservation Beweises und der Progress Teil wird voraussichtlich ohne größere Probleme funktionieren.

Äquivalenz Die Äquivalenz des logischen Typesystems mit dem constraintsgenerierenden Typesystems ist ein weiterer wichtiger Schritt, der noch erledigt werden muss.

Lösen Für das Lösen der Constraints ist eine Implementierung in Arbeit. Das Parsen des Quellcodes und das Generieren der Constraints ist komplett implementiert. Der Algorithmus zum Suchen einer Belegung für die Constraints ist in Arbeit.

3.2 Praxis

Interpreter Eine Implementierung eines JavaScript Interpreters in OCaml ist in erster Version fertig.

Datenbank Eine Datenbank, die JavaScript Quellcode aus dem Internet sammelt und eine statistische Auswertung über den Einsatz einzelner JavaScript Ausdrücke und Statements, wie z.B. das `with` Statement liefern soll, ist in Arbeit.

Literatur

1. ECMA International. ECMAScript language specification, 1998.
2. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.
3. Phillip Heidegger. Typbasierte Werkzeuge für Fehlersuche und Wartung von JavaScript Programmen, 2007. Deutschland, Universität Freiburg, Diplomarbeit.
4. Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.
5. Jürgen Schmidt. Online Banking fatal – Vom ausgezeichneten Webauftritt zum Security-Desaster. `http://www.heise.de/security/artikel/94451/`, Abruf: 31.8.2007, 13:00 Uhr, 2007.
6. Peter Thiemann. Towards a type system for analyzing javascript programs. In *European Symposium On Programming*, pages S. 408 – 422, 2005.
7. Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. 2007.