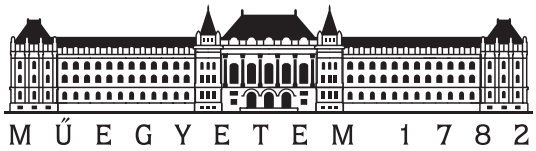

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Graph-Based Source Code Analysis of JavaScript Repositories

Master's Thesis

Author:

Dániel Stein

Supervisors:

Gábor Szárnyas
Ádám Lippai

2017.

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Stein Dániel*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2016. április 24.

Stein Dániel
hallgató

Abstract

Chapter 1

Introduction

Chapter 2

Background and Related Work

In this chapter I introduce the conceptual foundations and used technologies of my work. Here I discuss the building blocks required to create a static analyzer framework for JavaScript. I also enumerate a subset of similar systems and discuss related work.

2.1 JavaScript

2.1.1 Overview of the Last Few Years

2.1.2 The Future of JavaScript

2.2 Modeling

Modeling is a versatile concept, the word itself may refer to various topics. In the context of this thesis, by models I primarily mean data models. A data model —or sometimes called domain model— organizes the data elements, how they relate to each other and what actions one can perform on them.

2.2.1 Metamodels and Instance Models

Metamodeling is a methodology for the definition of modeling languages. A metamodel specifies the abstract syntax (structure) of a modeling language. [**scm**]

The metamodel contains the main concepts and relations of the domain specific language (DSL) and defines the possible structure of the instance models. To enrich the expressive power of the language, attributes are added to the concepts. By doing this, the language can be extended with predefined domains of data types (like integers, strings) that are supported by the metamodeling language. Additionally, some structural constraints might be specified with the elements like multiplicity.

Models describing a particular problem in the domain, called instance models, are defined using the elements of the metamodel.

2.3 Static Analysis

"The idea that computer software should be used to analyse source programs rather than compile them, has a history of at least 25 years." ([3])

We use program analysis to discover facts about it. Two basic automated analysis methods exist for source code analysis. When the source code is read and analyzed without evaluating the statements or executing it, we talk about static analysis. Analysis with statement evaluation or execution is known as dynamic analysis.

While JavaScript may not be, high-level language source codes are at least audited by the compiler. Thus the aim of a specific analysis should be discovering unwanted traits of the source in ways a generic compiler could not do, resulting in better code quality. These treats, *bugs* are usually perceptable while running the code. Another way to locate these bugs is to write and run tests, but it may not worth it, if there is a better solution.

2.3.1 Use Cases

The level of abstraction of the static analysis tools varies. Formatters make sure that the source code complies with the style guide. Linters check for stylistic and programming errors and warn at suspicious programming constructs. Formal verification, on the other hand, utilizes formal mathematical methods to prove or refute statements about the source code and its behavior.

For dynamic languages, it can be used for even more[1]: finding previously undefined property reads, invoke of non-functional variables [1] or dead code.

2.3.2 Advantages and Disadvantages

Since static analysis deliberately does not evaluate the source code, there are fundamental limitations to what problems it can discover. I consider the following advantages and disadvantages important.

Advantages

Input Data Is a By-Product of Compilers

No Need for Execution

[1] Non-exhaustive list

- no need for run environment
- no need for mocking, emulating

Disadvantages

Speed > Slower than Compilation

> Operationally, using static analysis to automatically find deep program bugs is about trading CPU time for the hardening of code. Because of the deep analysis performed by state-of-the-art static analysis tools, static analysis can be much slower than compilation.

> While the Clang Static Analyzer is being designed to be as fast and light-weight as possible, please do not expect it to be as fast as compiling a program (even with optimizations enabled). Some of the algorithms needed to find bugs require in the worst case exponential time.

Although, with a given limitation of granularity, in case of a source code modification, previous results can be reused. There is a possibility that only the modified —and other affected— parts need to be analyzed again. The incremental approach of static analysis may speed up the process by orders of magnitude [2].

False Positives > False Positives

> Static analysis is not perfect. It can falsely flag bugs in a program where the code behaves correctly. Because some code checks require more analysis precision than others, the frequency of false positives can vary widely between different checks. Our long-term goal is to have the analyzer have a low false positive rate for most code on all checks.

2.3.3 Typed JavaScript Derivations

2.3.4 Type Inference

> Type inference refers to the automatic deduction of the data type of an expression in a programming language. If some, but not all, type annotations are already present it is referred to as type reconstruction[citation needed]. The reverse operation of type inference is called type erasure.

> It is a feature present in some strongly statically typed languages. It is often characteristic of functional programming languages in general. Some languages that include type inference are Nim, ML, OCaml, F#, Haskell, Scala, Go, D, Clean, Opa, Rust, Swift, Visual Basic (starting with version 9.0), C# (starting with version 3.0) and C++11. The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

Flow

> Flow, a new open-source static type checker for JavaScript. Flow adds static typing to JavaScript to improve developer productivity and code quality. In particular, static typing offers benefits like early error checking, which helps you avoid certain kinds of runtime failures, and code intelligence, which aids code maintenance, navigation, transformation, and optimization.

> Flow can infer the type of most things within a file, so you don't always have to annotate every function and variable to get typechecking to work. However, even if Flow can infer a type, you can still add annotations to be explicit. The only time that Flow strictly requires an annotation is when a variable/function/class is exported from a module (defined in one file and used in another). > > In our final example, 05_DynamicCode, we haven't annotated the function, but we are passing in two different types of arguments: > > In this case, Flow detects that the second time the function is called (with a number), the length property will fail: > > Flow is smart enough to detect that this conditional check is sufficient to avoid any potential failures at run time, and will give you a clean bill of health.

> Since types are not part of the JavaScript specification, we need to strip them out before sending the file to the user.

> Making previously-untyped code typecheck with Flow may take some time and work - and sometimes it may not be worth the effort in the short term. Flow supports interface files so you can use libraries in a typed way without having to run Flow on them at all. If your project just depends on third party libraries, check out our guide on using Flow with external dependencies and consider using an interface file for the libraries. > >Why is typechecking existing code so hard? Libraries not written with types in mind often contain complex, highly dynamic code that confuses analyzers such as Flow. The code may also have been written in a style that Flow deliberately chooses not to support in order to give the programmer more help. Some typical examples are: > > Operations on primitive values: While JavaScript allows operations such as true + 3, Flow considers it a type error. This is by design, and is done to provide the programmer with more safety. While that's easily avoided for new code, it can sometimes be a lot of effort to eliminate such patterns from existing code. > Nullability: Flow protects you against accessing properties on null by tracking null or undefined values throughout the program. In large existing codebases, though, this can require inserting some extra null checks in places where a value appears like it may be null, but actually isn't at runtime. > > It is typically a much larger effort, and requires much more programmer annotation, to get such code to typecheck. On the other hand, if you own a library and would like to benefit from Flow typechecking within the library itself, this guide is for you.

> When you start Flow, it performs an initial analysis of all the files in your codebase and stores the results in a persistent server. When you save a file, Flow incrementally rechecks the changes in the background. > > Both the initial analysis and recheck are heavily optimized for

performance, which preserves the fast feedback of developing plain JavaScript. > > Flow uses control flow analysis to deeply understand your code to find errors that other type systems can't. Flow is designed to find errors and we take soundness seriously. > > For example, Flow tracks null values which may propagate unintentionally through code and eventually cause a runtime error. Flow's path sensitive analysis can uncover bugs like this, even through layers of indirection in the program's control flow

> Facebook loves JavaScript; it's fast, it's expressive, and it runs everywhere, which makes it a great language for building products. At the same time, the lack of static typing often slows developers down. Bugs are hard to find (e.g., crashes are often far away from the root cause), and code maintenance is a nightmare (e.g., refactoring is risky without complete knowledge of dependencies). Flow improves speed and efficiency so developers can be more productive while using JavaScript.

> But layering a static type system onto JavaScript is not trivial. JavaScript's building blocks are extremely expressive, and simple type systems do not suffice to precisely model their semantics. To seamlessly work with several common JavaScript idioms, Flow employs the kind of data-flow and control-flow analysis that compilers typically perform to extract semantic information from code. It then uses this information for type inference, building on advanced techniques in type theory. Of course, designing a powerful static analysis is not sufficient — JavaScript codebases can be huge, so type checking has to be blazing fast as not to disrupt the developer's edit-run cycle. Flow performs its analysis modularly, guided by types at module boundaries. This enables an aggressively parallel and incremental type checking architecture, similar to Hack. This allows type checking to appear instantaneous, even on millions of lines of code. > > Flow's type checking is opt-in — you do not need to type check all your code at once. However, underlying the design of Flow is the assumption that most JavaScript code is implicitly statically typed; even though types may not appear anywhere in the code, they are in the developer's mind as a way to reason about the correctness of the code. Flow infers those types automatically wherever possible, which means that it can find type errors without needing any changes to the code at all. On the other hand, some JavaScript code, especially frameworks, make heavy use of reflection that is often hard to reason about statically. For such inherently dynamic code, type checking would be too imprecise, so Flow provides a simple way to explicitly trust such code and move on. This design is validated by our huge JavaScript codebase at Facebook: Most of our code falls in the implicitly statically typed category, where developers can check their code for type errors without having to explicitly annotate that code with types.

- `http://flowtype.org/`
- `https://github.com/facebook/flow`
- `https://code.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-fo`

TAJS

- http://www.brics.dk/TAJS/
- https://github.com/cs-au-dk/TAJS
- http://www.srl.inf.ethz.ch/workshop2014/eth-moeller.pdf

JSNice – Programming Tools with Big Data and Conditional Random Fields (ETHZ)

- http://www.srl.inf.ethz.ch/jsnice.php
- http://www.srl.inf.ethz.ch/papers/jsnice15.pdf
- http://www.jsnice.org/
- https://github.com/eth-srl/Nice2Predict

Mozilla DoctorJS

- http://rfrn.org/~shu/drafts/ti.pdf (2012)
- https://wiki.mozilla.org/TypeInference
- https://github.com/dimvar/doctorjs
- https://github.com/mozilla/doctorjs

Tern

- http://marijnhaverbeke.nl/blog/tern.html
- http://ternjs.net/doc/manual.html#infer
- http://ternjs.net/doc/manual.html#typedef
- https://github.com/angelozerr/tern.java
- https://github.com/ternjs/tern/blob/master/lib/infer.js

JS0 – PhD (2006) @ University of London

- http://dev.pubs.doc.ic.ac.uk/chrisandersonphd/chrisandersonphd.pdf

Infernu

- `https://github.com/sinelaw/infernu`

University of California

- `https://www.cs.ucsb.edu/~benh/research/papers/kashyap13type.pdf(2013)`

2.4 Graph Databases

2.4.1 Adequate Database Solutions

- Neo4j - Titan - Dragon (?) - Cayley...

2.4.2 Query Languages and Evaluation Strategies

There are numerous strategies to define and execute a query. Queries can be expressed in imperative programming languages over a data access interface such as the Eclipse Modeling Framework (EMF, see emf), or with declarative languages, processed by a query framework, such as OCL [**OCL**] or EMF-IncQuery [**IncQuery**].

Pattern matching, one of the various methods to retrieve data from a model is what we base our approach on. Following [**csmr**], we define graph patterns and discuss how they are used for querying.

Graph patterns are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. The formalism is useful for various purposes in model-driven development, such as defining model transformation rules or defining general purpose model queries including model validation constraints. A graph pattern consists of structural constraints prescribing the interconnection between nodes and edges of a given type.

Graph patterns are extended with expressions to define attribute constraints and pattern composition to reuse existing patterns. The called pattern is used as an additional set of constraints to meet, except if it is formed as negative application condition (NAC) describing cases when the original pattern does not hold.

Pattern parameters are a subset of nodes and attributes interfacing the model elements interesting from the perspective of the pattern user.

A match of a pattern is a tuple of pattern parameters that fulfill all the following conditions:

1. have the same structure as the pattern,
2. satisfy all structural and attribute constraints,

3. and does not satisfy any NAC.

When evaluating the results of a graph pattern, any subset of the parameters can be bound to model elements or attribute values that the pattern matcher will handle as additional constraints. This allows re-using the same pattern in different scenarios, such as checking whether a set of model elements fulfill a pattern, or list all matches of the model.

SPARQL SPARQL (recursive acronym for SPARQL Protocol and RDF Query Language) is an RDF query language. (RDF is discussed in detain in RDF.) SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs. [**W3C-SPARQL**, **Harris:13:SQL**]

For an example SPARQL query, see sparql-query.

Chapter 3

Overview of the Approach

Chapter 4

Elaboration of the Workflow

Chapter 5

Evaluation of the Prototype

Chapter 6

Future Vision

Chapter 7

Conclusions

Acknowledgments

List of Figures

List of Tables

References

[1] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type analysis for JavaScript”. In: *Static Analysis*. Springer, 2009, pp. 238–255. URL: http://link.springer.com/chapter/10.1007/978-3-642-03237-0_17 (visited on 04/20/2016).

[2] Dániel Stein. “Incremental Static Analysis of Large Source Code Repositories”. Bachelor’s thesis. Budapest University of Technology and Economics, 2014.

[3] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh. “Industrial perspective on static analysis”. In: *Software Engineering Journal* 10.2 (1995), p. 69. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.4760&rep=rep1&type=pdf (visited on 04/22/2016).

Appendix