Pointer Analysis for JavaScript Programming Tools

Asger Feldthaus

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Pointer Analysis for JavaScript Programming Tools

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Asger Feldthaus
October 31, 2014
Abstract

Tools that can assist the programmer with tasks, such as, refactoring or code navigation, have proven popular for Java, C#, and other programming languages. JavaScript is a widely used programming language, and its users could likewise benefit from such tools, but the dynamic nature of the language is an obstacle for the development of these. Because of this, tools for JavaScript have long remained ineffective compared to those for many other programming languages.

Static pointer analysis can provide a foundation for more powerful tools, although the design of this analysis is itself a complicated endeavor. In this work, we explore techniques for performing pointer analysis of JavaScript programs, and we find novel applications of these techniques. In particular, we demonstrate how these can be used for code navigation, automatic refactoring, semi-automatic refactoring of incomplete programs, and checking of type interfaces.

Subset-based pointer analysis appears to be the predominant approach to JavaScript analysis, and we survey some techniques for flow-insensitive, subset-based analysis and relate these to a range of existing implementations. Recognizing that the subset-based approach is hard to scale, we also investigate equality-based pointer analysis for JavaScript, which, perhaps undeservingly, has received much less attention from the JavaScript analysis community.

We show that a flow-insensitive, subset-based analysis can provide the foundation for powerful refactoring tools, while an equality-based analysis can provide for tools that are less powerful in theory, but more practical for use under real-world conditions. We also point out some opportunities for future work in both areas, motivated by our successes and difficulties with the two techniques.
Værktøjer, der kan hjælpe programmeren med opgaver såsom refaktorisering eller kodenavigering, har vist sig populære til brug i Java, C# og andre programmeringssprog. JavaScript er et udbredt programmeringssprog, og dets brugere kunne ligeledes drage fordel af sådanne værktøjer, men sprogets dynamiske karakter er en hindring for udviklingen af disse. Dette har medført en ringere standard blandt JavaScript-værktøjer sammenlignet med værktøjer til mange andre programmeringssprog.

Statisk pointeranalyse kan give et fundament til at bygge mere effektive værktøjer, men udformningen af sådan en analyse er i sig selv en vanskelig affære. I denne afhandling undersøger vi teknikker til pointeranalyse af JavaScript-programmer, og vi finder nye anvendelser af disse. Vi viser konkret hvordan forskellige analyser kan bruges til kodenavigering, automatisk refaktorisering, delvist automatisk refaktorisering af ukomplette programmer samt kontrol af typespecifikationer.

Mængdebaseret pointeranalyse lader til at være den fremherskende tilgang til JavaScript analyse, og vi beskriver nogle teknikker til flow-insensitiv mængdebaseret analyse, og sammenholder med en række eksisterende implementationer. I erkendelse af at den mængdebaserede analyse er svær at skalere, undersøger vi også ækvivalensbaseret analyse, hvilket, måske ufortjent, har fået langt mindre opmærksomhed indenfor forskningen af JavaScript analyse.

Vi påviser at en mængdebaseret analyse kan fungere som fundament i kraftfulde refaktorisingsværktøjer, hvorimod den ækvivalensbaserede analyse giver anledning til værktøjer der er mindre kraftfulde i teorien, men er mere praktiske til anvendelse under virkelighedens omstændigheder. Vi udpeger desuden nogle muligheder for fremtidigt arbejde indenfor begge områder, motiveret af vores egne succeser og vanskeligheder med de to teknikker.
I would like to thank my supervisor, Anders Møller, for excellent guidance, inspiring discussions, and sound judgment calls during my PhD studies. Anders supported my pursuit of the occasional unorthodox idea, but also pulled me back to reality in cases where things were leading nowhere. Anders insisted on understanding my work, despite the cryptic materials I sometimes provided him, and I would not have had half as much fun, or learned half as much, had he not shared my enthusiasm for many an ambitious project.

I would also like to thank my office mate, Magnus Madsen, for several years in good company, and for many exciting discussions about programming languages and static analysis. Magnus repeatedly opened my eyes to research I ought to know about, books I ought to read, and videos of flying lawn mowers that I will not soon forget.

I also thank the other members and former members of the programming language group for creating a positive and productive atmosphere. In particular, I thank Esben Andreasen and Casper Svendsen for their helpful feedback over the years and for a fun trip to Marktoberdorf. I thank Jakob G. Thomsen for teaching me humility at the foosball table. I also thank Ian Zerny, Jan Middgaard, and Olivier Danvy for sharing their comprehensive knowledge of formal languages and abstract interpretation, without which I feel I might have assumed a narrow perspective on my own field of research.

For an enjoyable internship at IBM Research, I would like to thank Manu Sridharan, Julian Dolby, Frank Tip, and Max Schäfer. Together with Max, I have had particularly good fun with joint implementation work, and an enlightening exchange of ideas. As my fellow interns, Avneesh, Liana, Mauro, Waldemar, Marc, Puja, Ioan, and Sihong also deserve gratitude for making my stay in New York a fun and exciting episode of my life, and for expanding my knowledge of computer science well beyond my own field.

I would like to thank Yannis Smaragdakis for his extensive hospitality during my short stay in Athens. I enjoyed spending time with Yannis and his students, and under different circumstances, I would have liked to do more work together.

I also thank Kevin Millikin and the rest of the Google Aarhus team for an exciting internship. The disciplined but happy coding culture at Google provided a welcome break from academia, and I learned much from working with such brilliant individuals.

I would also like to thank the staff at Aarhus University, in particular, Ann Eg Mølhave for always shielding me against bureaucratic tyranny and administrative paperwork.
Lastly, I would like to thank my parents. They acknowledged my passion for programming at an early age. They showed me the bare bones of trees and loops and memory, and mostly let me learn the rest on my own.

Asger Feldthaus,
Aarhus, October 31, 2014.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Resumé</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
</tbody>
</table>

## Overview

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 A Brief Look at JavaScript</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Tools for JavaScript Development</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Static Analysis</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Overview of this Dissertation</td>
<td>11</td>
</tr>
</tbody>
</table>

## Subset-based Pointer Analysis

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Constraint Systems</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Andersen-style Analysis</td>
<td>14</td>
</tr>
<tr>
<td>2.3 JavaScript Mechanics</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Conclusion</td>
<td>39</td>
</tr>
</tbody>
</table>

## Equality-based Pointer Analysis

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Steensgaard-style Analysis</td>
<td>41</td>
</tr>
<tr>
<td>3.2 Union-Find</td>
<td>43</td>
</tr>
<tr>
<td>3.3 JavaScript Mechanics</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>48</td>
</tr>
</tbody>
</table>

## Other Approaches

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Pushdown-based Analysis</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Dynamic Analysis</td>
<td>50</td>
</tr>
<tr>
<td>4.3 Type Systems for JavaScript</td>
<td>52</td>
</tr>
</tbody>
</table>
Contents

II Publications 53

5 Tool-supported Refactoring for JavaScript 55
  5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
  5.2 Motivating Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
  5.3 A Framework for Refactoring with Pointer Analysis . . . . . . . . . . . . . 66
  5.4 Specifications of Three Refactorings . . . . . . . . . . . . . . . . . . . . . . 72
  5.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
  5.6 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
  5.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
  5.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

6 Approximate Call Graphs for JavaScript IDE Services 87
  6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
  6.2 Motivating Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
  6.3 Analysis Formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
  6.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
  6.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
  6.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

7 Semi-Automatic Rename Refactoring for JavaScript 107
  7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
  7.2 Motivating Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
  7.3 Finding Related Identifier Tokens . . . . . . . . . . . . . . . . . . . . . . . . 113
  7.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
  7.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
  7.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

8 Checking Correctness of TypeScript Interfaces for JavaScript Libraries 133
  8.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
  8.2 The TypeScript Declaration Language . . . . . . . . . . . . . . . . . . . . . . 137
  8.3 Library Initialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
  8.4 Type Checking of Heap Snapshots . . . . . . . . . . . . . . . . . . . . . . . 142
  8.5 Static Analysis of Library Functions . . . . . . . . . . . . . . . . . . . . . . 146
  8.6 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
  8.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
  8.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160

Bibliography 161
Part I

Overview
Chapter 1

Introduction

Pointer analysis techniques can substantially improve programming tools for JavaScript, despite the obstacles posed by the dynamic nature of the language. To demonstrate, we shall explore various pointer analysis techniques for JavaScript and show how these can be used in tools for automatic refactoring, semi-automatic refactoring for incomplete programs, code navigation, and checking of type interfaces.

In practice, we find that the manual effort required to perform rename refactoring can roughly be reduced to half, code navigation can be provided with very little cost and complexity, and a large number of type mismatches can be detected automatically in type interfaces between JavaScript and TypeScript. Testing the limits of what is feasible, we also show that for complete programs of limited complexity, various types of refactoring can be fully automated, and with effective safeguards against incorrect refactorings.

For several statically typed languages, such as Java and C#, similar and more powerful tools already exist, or are made entirely unnecessary by the static type system. Although a sizable gap remains between the two worlds, our results show that there are practical ways to carry over some of these benefits, traditionally reserved for statically typed languages, into the dynamically typed world.

In the rest of this chapter we will give a brief introduction to JavaScript and the nature of the tools we implement; we then provide some background material on static analysis. In remainder of the first part of this thesis, we survey techniques for static analysis of JavaScript, while the second part consists of publications that focus on the tools.

1.1 A Brief Look at JavaScript

JavaScript is a dynamically typed, imperative language. It has no class, module, or import declarations, which are present in some other dynamically typed languages, such as Python. Instead, such features are often mimicked using its operational features.

To analyze JavaScript code and build effective tools for it, we must consider both the mechanics of the language as well as commonly used coding style and idioms. In this section, we will showcase some patterns that are common in JavaScript and explain their underlying mechanics. We suffice with a cursory explanation for now – the technical details will be revisited on-demand as they become relevant throughout the dissertation.
1. Introduction

JavaScript does not have classes like in Java, C#, and Python. However, class hierarchies can be emulated using a combination of prototype-based inheritance and the mechanics for the this argument. For example, in the following we define a string buffer class using this idiom:

```javascript
function StringBuffer() {
  this.array = []
}
StringBuffer.prototype.append = function(x) {
  this.array.push(x)
}
StringBuffer.prototype.toString = function() {
  return this.array.join('')
}
```

Although the StringBuffer function is technically an ordinary JavaScript function, it can be thought of as a constructor for the class. The two functions created on line 4 and 7 are also ordinary functions, stored in a property on StringBuffer.prototype, but they can be thought of as methods of the class. The following code demonstrates its use:

```javascript
var sb = new StringBuffer()
sb.append('foo')
sb.append('bar')
console.log(sb.toString()) // prints 'foobar'
```

Although this code has the "look and feel" of classes and methods in Java and C#, the underlying mechanics are different.

The constructor call on line 10 creates a new object, which inherits the properties from StringBuffer.prototype. This object can be thought of an as instance of the string buffer class. The instance object is passed as the value of this to the StringBuffer function on line 13. An empty array is then created on line 2 and stored in the array property of the string buffer instance. When the constructor returns, the instance object gets assigned to the variable sb.

Two method calls of form sb.append(...) are then performed on line 11-12. Functions are first-class values, and the expression sb.append evaluates to the append function from line 4, because sb inherits from StringBuffer.prototype. The value of sb is passed as this because it is syntactically the base expression in sb.append. On line 13 the toString method is invoked by similar mechanics, and its result printed to the console.

The following example displays a couple of other idioms:

```javascript
function Vector(x, y) {
  this.x = x || 0
  this.y = y || 0
}
Vector.prototype.add = function(v, y) {
  if (v instanceof Vector) {
    this.x += v.x
    this.y += v.y
  } else {
    this.x += v || 0
    this.y += y || 0
  }
}
var a = new Vector(); // x: 0, y: 0
a.add(new Vector(1,2)); // x: 1, y: 2
a.add(10, 30); // x: 11, y: 32
```

A logical or expression, x || y, evaluates x, and if the result is a true-like value, that value is returned, otherwise y is evaluated and its value is returned. In addition to being used
as a traditional lazy boolean operator, as in Java, it is often used to replace a "missing" value with a meaningful default, such as zero. On line 27 the Vector function is invoked with fewer arguments than it declares; the excess parameters are filled with the value undefined, which is considered a false-like value. The Vector constructor on line 14 fills in zeros as defaults (instead of undefined) if the parameters are not provided.

The add method on line 18 uses an instanceof expression to distinguish the case where a vector object is passed as argument, and the case where one or two numbers are passed as arguments. Since the language does not support overloading, it must be implemented manually using this kind of runtime check.

### 1.2 Tools for JavaScript Development

We shall consider three types of tools that can be powered by static analysis: refactoring, code navigation, and checking correctness of TypeScript interfaces.

#### 1.2.1 Refactoring

Refactoring is a process of transforming a program while preserving its behavior, typically carried out to improve the quality of the code. Refactoring tools automate certain predefined types of refactorings, such as renaming a variable or inlining a function. Since their incarnation in the early 1990s refactoring tools have become popular in modern software development, particularly for statically typed languages, such as Java and C#.

For instance, the Eclipse IDE can automatically rename a given field or method in a Java application, ensuring that all relevant references are updated to use the new name. Such functionality is hard to implement for JavaScript due to its highly dynamic nature, and a large part of our work addresses this particular problem.

To demonstrate, consider the following example of how one might rename a property array to strings:

```javascript
function StringBuffer() {
    this.array = []
}
StringBuffer.prototype.append = function(x) {
    this.array.push(x)
}
StringBuffer.prototype.toString = function() {
    return this.array.join('')
}
```

```javascript
function StringBuffer() {
    this.strings = []
}
StringBuffer.prototype.append = function(x) {
    this.strings.push(x)
}
StringBuffer.prototype.toString = function() {
    return this.strings.join('')
}
```

The identifiers being renamed above are not bound by lexical scoping, hence the transformation can generally not be contained locally within some syntactic scope. If there are more methods added to StringBuffer elsewhere, they might no longer work if they were
1. Introduction

not updated during the refactoring. Moreover, it is not always clear if any given identifier
token named `array` should be updated during a renaming. For instance, suppose the
following was part of the same application:

```javascript
function BinaryHeap() {
  this.array = []
}
BinaryHeap.prototype.insert = function(key, value) {
  var i = this.array.length
  while (...) { ... } // omitted
  this.array[i] = { key: key, value: value }
}
```

The `BinaryHeap` class also uses a property named `array`, but it is entirely possible that
this is by coincidence, and that there is no intended relationship between the properties. It
may even be that the renaming was deliberately meant to disambiguate the two property
names. In this particular case it is fairly evident, at least in the eyes of a programmer,
that these property names are unrelated. But in some cases, there is no local indication
of whether a token should be renamed, as in the following function:

```javascript
function getFirst(x) {
  return x.array[0];
}
```

Without further information, there is no way to tell if `x` is intended to be a `StringBuffer`
or a `BinaryHeap`. It may also be that the function is polymorphic, in that it is intended
to work for both types of objects. In the latter case, a consistent renaming must update
`array` in `getFirst` and in `BinaryHeap` and its methods. Chapters 5 and 7 explore two
different approaches to this problem:

Chapter 5 explores the use of whole-program pointer analysis to automate refactoring.
By running a pointer analysis on the input, we can determine how different parts of the
code are actually used. The above dilemma is then resolved statically based on whether
the code contains a call to `getFirst` taking a `StringBuffer` and/or a `BinaryHeap` object
as argument. By checking certain preconditions, we can verify if the program has the
same behavior after the refactoring. The technique is fully automated, and the technique
can also be used for other types of refactoring, in particular, extracting a module and
encapsulating a property. The design of the underlying pointer analysis is discussed in
detail in Chapter 2.

The approach is testimony to the usefulness of static pointer information; its main
drawback is that reliable pointer information can only be obtained for complete pro-
grams. For instance, if the above code snippets were part of a library, the library itself
may not contain any calls to `getFirst`. If the library is analyzed in isolation (without
a comprehensive test suite or a client), pointer information does not indicate how the
function is intended to be used.

In Chapter 7 we recognize that a renaming problem for partial programs is not purely
a computational one: we lack some input about the intended use of various code fragments.
Type annotations serve this purpose in Java and C#, so techniques from that world cannot
be applied to JavaScript. We propose a semi-automatic renaming technique which resolves
the above dilemma by involving the user in the renaming process. In this approach, we use
a static analysis to determine a minimal set of yes/no questions to ask the programmer in
order to uniquely determine which identifier tokens to rename. Importantly, the analysis
does not rely having the whole program available (although obviously it cannot update
source code that is not present when applied).
1.2. Tools for JavaScript Development

This puts it in league with text-based search-and-replace tools, which essentially prompt the user for every occurrence of the renamed identifier, asking whether it should be replaced. Compared to such tools, the use of static analysis reduces the amount of effort required from the user by deducing the "obvious" answers automatically.

We developed plugins for both approaches, adding JavaScript refactoring functionality to the Eclipse IDE and the Brackets IDE.

1.2.2 Code navigation

Another hallmark of a modern IDE is its ability to jump to the declaration of a function or show the parameters it expects in a pop-up. Given a call expression `x.insert()`, the user might wish to jump to the declaration of `BinaryHeap.prototype.insert` (in the previous example), or simply open a pop-up to remind that it takes two parameters, `key` and `value`. Such tools are realizable if a call graph can be obtained for the program, indicating that the call expression can invoke the function in `BinaryHeap.prototype.insert`.

In Chapter 6, we experiment with a simple heuristic that approximates the call graph of JavaScript programs by reducing the problem to graph reachability. The resulting call graph is not guaranteed to be correct (nor is it in practice), but empirical evidence suggests that it is more accurate than one might expect, given that the algorithm completely disregards several commonly used JavaScript mechanics. It can thus be useful for applications that require high performance but can tolerate the lack of accuracy.

1.2.3 Checking correctness of TypeScript interfaces

The benefits of static types systems have led to various attempts at adding types to JavaScript. A popular variant is Microsoft’s TypeScript language, which extends JavaScript with explicit class declarations, modules, and optional types. Optional types do not guarantee type safety, but can facilitate productivity tools, such as refactoring and code navigation, for applications written in TypeScript. TypeScript can be translated to JavaScript by desugaring the new syntax and erasing all type annotations. The directness of this translation means TypeScript and JavaScript code can interact without the use of a foreign-function interface at runtime. This effectively places TypeScript and JavaScript in the same ecosystem, since libraries written in one language can be used in the other.

This provides an alternative for developers who wish to target the JavaScript platform and make use of existing libraries for it. The use of static types is not without cost, however. In order to type-check a TypeScript application that uses a JavaScript library, the interface to the JavaScript library must be declared in TypeScript code, in an interface definition. Unfortunately, these must be written by hand, and often contain bugs. Bugs in interface definitions degrade the quality of tools that depend on these types, for example, code navigation might display misleading information about function parameters, or the type checker might reject code that it would otherwise have accepted.

In Chapter 8, we use static analysis to detect mismatches between TypeScript interface definitions and the JavaScript libraries whose API they are intended to describe. We published the tool TSCheck based on this technique and found over a hundred type mismatches in TypeScript interfaces.

1. Introduction

1.3 Static Analysis

A large part of this dissertation is on the topic of static analysis, since every tool we propose use it in some form, and it is in all cases the most complex component of the tool.

Static analysis is a form of program analysis, which in general, is the process of determining some facts about the behavior of a program. An algorithm for performing program analysis is itself referred to as an analysis. A static analysis is one that performs its analysis without executing the program, as opposed to a dynamic analysis, which learns about the program by observing and possibly influencing its execution.

The purpose of a static analysis is to answer certain questions about the program, such as whether its output can be a negative number, or whether an exception may be thrown during its execution. It is well established that any non-trivial question regarding the output, or what can or cannot happen during execution, is undecidable. As a consequence, an analysis must occasionally answer with partial information, such as 'the output is zero or negative' in cases where it cannot actually output zero, and in some cases even provide downright information-less answers such as "an exception may or may not be thrown".

There are three fundamental metrics for the quality of any static analysis that attempts to decide if the input has some property $X$:

**Soundness:** If the analysis says $X$ holds, is this true (sound) or false (unsound)? This could also be called correctness or sometimes recall, but soundness is the standard term in the static analysis community. An analysis is said to be sound if it is guaranteed to only give sound answers.

**Precision:** If $X$ holds, can the analysis conclude this (precise), or does it conservatively say that "$X$ may or may not hold" (imprecise)?

**Efficiency:** How fast is the analysis? How much memory does it require? Does it scale to large programs? Is it even guaranteed to terminate?

We use the term accurate as an aggregate of both soundness and precision, in the sense that a highly accurate analysis has a high degree of soundness and precision.

The terms 'may' and 'must' have a strict meaning in the context of static analysis. A fact such as "$x$ may be zero" should be read as "we cannot prove that $x$ is not zero". In other words, such a fact represents our lack of knowledge about $x$, and does not guarantee that it is actually possible for $x$ to be zero. Calling such a statement a fact is therefore a misnomer, but this terminology is commonly used. A fact such as "$x$ must be zero" is less subtle; it can be read as "we have proven that $x$ is zero" and thus represents actual knowledge about $x$.

When discussing the design of static analysis, we often refer to the client, as whatever mechanism is going to make use of the data produced by the analysis. Depending on the client, some design choices can be justified while they would be meaningless for other clients.

1.3.1 Pointer Analysis

Pointer analysis is an important and well-studied class of static analysis [44]. A pointer analysis aims to determine which pointers refer to the same storage locations, which in turn is important for reasoning about dataflow problems, that is, how the values of
variables and expressions influence each other in a program. Pointer analyses provide a heap abstraction for reasoning about the flow of values through heap-allocated storage.

The are significant differences between pointer analysis for memory-safe languages like JavaScript, and unsafe languages like C. In C programs, pointers may reference stack variables or a specific field of an object, and pointers can be manipulated using arithmetic. In memory-safe languages like JavaScript, however, pointers can only refer to objects and pointer arithmetic is prohibited. We shall focus the aspects of pointer analysis that are relevant for JavaScript, which excludes the complexity of unsafe pointers.

There are several dimensions commonly used to classify pointer analyses:

**Flow-sensitivity:** Is pointer information obtained separately for each program point, or is a single heap abstraction used to model the state of the heap at all times? Flow-sensitive analyses trade some efficiency in favor of greater precision.

**Context-sensitivity:** Does the abstract state in a function depend on the calling context, or can values from different calling contexts spuriously influence each other? In principle, context-sensitive analyses trade efficiency for greater precision, although in practice, a context-sensitive analysis is sometimes faster than an insensitive one.

**Directional:** Are assignments treated as directional or bidirectional? Analyses that use bidirectional assignments can represent aliasing with a fast union-find data structure, and thus trade significant precision in favor of greater efficiency.

**Whole program:** Does the analysis depend on the whole program, or can meaningful results be obtained for partial or incomplete programs? Accurate pointer analysis for partial programs is difficult when the unseen parts of the program may establish aliasing. We use the term *partial programs* for programs that interact with some components that are not available (e.g. a library may interact with some unseen client code), and the term *incomplete* programs for programs that are under development and thus may internally be missing some code fragments.

In this dissertation we shall explore three common classes of pointer analysis and their use for programming tools:

**Subset-based:** The heap is represented using a finite set of abstract objects. Assignments are directional and the analysis can be either flow-sensitive or insensitive. The flow-insensitive variant is also known as Andersen-style [5], inclusion-based, or points-to analysis. We discuss this style in Chapter 2 and use it for refactoring in Chapter 5.

**Equality-based:** Aliasing is represented as an equivalence relation. Assignments are bidirectional and the analysis is traditionally flow-insensitive. This type of analysis is also known as Steensgaard-style [82] or unification-based analysis. We discuss this type of analysis in Chapter 3 and use it for renaming in Chapter 7 and for checking type interfaces in Chapter 8.

**Field-based:** Different objects are not distinguished, only their fields are. Since fields cannot be aliased, this greatly simplifies the analysis. Such an analysis is equivalent to a subset-based analysis with a single abstract object, but the implementation and relevant design choices differ significantly. We describe the design of a simple, but surprisingly accurate, field-based analysis for code navigation in Chapter 6.
1. Introduction

In the design of pointer analysis, the trade-off between precision and efficiency is blurred by a phenomenon where imprecision causes loss of performance. It is not uncommon for a very precise analysis to run faster than an imprecise one for many inputs. In a subset-based analysis, the time and space cost is (roughly) proportional to the number of facts it concedes that may or may not hold. While a precise analysis explores a more fine-grained space of facts (thus may concede more facts in the worst case), it may easily be the case that it concedes fewer facts in practice due to its higher precision.

Flow-insensitivity typically does improve efficiency in practice, despite the phenomenon mentioned above. This is not the case for context-insensitivity, however. A completely context-insensitive analysis often converges towards its worst-case complexity (extremely expensive) whereas using a limited amount of context-sensitivity allows it to terminate quickly. On the other hand, using an overly aggressive heuristic for context-sensitivity will typically cause a blow up in time and space usage even for small inputs.

1.3.2 Existing JavaScript Pointer Analyses

A number of pointer analyses for JavaScript have been proposed and implemented. The following ones analyze a reasonably comprehensive part of the language:

**TAJS:** A flow-sensitive, subset-based pointer analysis for JavaScript [46]. TAJS aims to analyze the whole JavaScript language soundly, even eval [49], it has a comprehensive model of the HTML DOM [48], and it has been used as a platform for research toward analyzing jQuery applications [7]. Its main client has been bug-detection and proving the absence of some common types of runtime errors.

**WALA:** A multi-language, flow-insensitive, subset-based analysis with support for a large part of JavaScript. WALA has also been used for jQuery analysis [75, 80]. To name a few clients, Guarnieri et al. [38] used it for taint analysis to detect vulnerable JavaScript code; Wei and Ryder [91] combined it with a dynamic analysis to perform precise taint analysis for web pages; Schäfer et al. [76] demonstrated it can be used for code completion if combined dynamically generated with library models.

**Gatekeeper:** A flow-insensitive, subset-based analysis for a subset of JavaScript [36]. It has been used to detect malicious code, coupled with a runtime instrumentation to guard the mechanics that could not be analyzed statically. More recently, Madsen et al. [59] adapted it to perform code completion for partial programs.

**JSRefactor:** A flow-insensitive, subset-based analysis underlying the fully automated refactoring technique in Chapter 5. Designed to analyze all mechanics of JavaScript except eval.

**DoctorJS:** A flow-sensitive and highly context-sensitive analysis for JavaScript based on CFA2 [87] (also see Section 4.1). This has been used for inferring function types and detecting API usage in Firefox extensions.

**SAFE:** An intermediate representation for JavaScript with a big-step operational semantics [55]. It has been used in a static analyzer to detect improper use of browser API on web pages [10].

---

3 jQuery is difficult to analyze, but is of particular interest due to its large market share among JavaScript libraries [90].
1.4. Overview of this Dissertation

**JSAI:** A flow-sensitive dataflow analysis for JavaScript with an emphasis on configurability and verifiable correctness [52]. Kashyap et al. [51] used JSAI as part of a tool to detect malicious code in Firefox extensions.

**TSCheck:** An equality-based analysis that reasons about pointers together with type information. In Chapter 8 we use it to detect errors in a TypeScript interface file.

All of the static analyses above depend on the whole program, with the exception of TSCheck, which uses a type interface to model the client-side of a library.

Some of these analyzers lack sound support for certain language mechanics. Type coercions, property getters and setters, `with` statements, and `eval` are among the usual suspects. It is also possible that some combination of mechanics is unsupported. This can be simple combinations, such as dynamic property reads that access a prototype, or more obscure scenarios, such as `this` escaping out of an implicit call to `toString` using an exception. Many of such shortcomings have limited impact in practice, unless one intends to analyze adversarial code. For productivity tools, such as refactoring or bug-finding, the programmer has no interest in deliberately targeting the weak points of the analysis.

The above list of JavaScript analyzers is not exhaustive. For example, Jang et al. [45] propose a points-to analysis for a subset of JavaScript without the prototype mechanic (and several other mechanics); we consider this too restrictive to warrant a detailed comparison with the other analyzers. The analysis used for semi-automatic renaming in Chapter 7 does reason about pointers to some degree, but is too specialized to its application domain for a comparison to be relevant.

1.4 Overview of this Dissertation

The following three chapters discuss various techniques for static analysis and the last four chapters discuss the design and implementation of programming tools for JavaScript:

**Chapter 2:** We introduce the basic concepts of subset-based analysis and discuss some techniques for handling the mechanics of JavaScript in this setting.

**Chapter 3:** We introduce the basic concepts of equality-based analysis and discuss, more briefly, ways in which this can be adapted to JavaScript.

**Chapter 4:** We discuss the potential for using other types of analyses as the foundation for programming tools.

**Chapter 5:** We demonstrate how a subset-based analysis can be used as the foundation for fully automated refactoring. This chapter was published at OOPSLA’11 [27].

**Chapter 6:** We demonstrate how a simple field-based analysis can be used for code navigation and a few other IDE services. This chapter was published at ICSE’13 [23].

**Chapter 7:** We demonstrate how an under-approximate equality-based analysis can be used for semi-automatic renaming. This chapter was published at OOPSLA’13 [24].

**Chapter 8:** We demonstrate how a snapshotting technique, structural type checking, and an equality-based analysis can be combined to find bugs in TypeScript interfaces. This chapter was published at OOPSLA’14 [25].
Chapter 2

Subset-based Pointer Analysis

Subset-based pointer analyses reason about the heap by dividing the potentially infinite space of objects into a finite set of components, called abstract objects or labels. The allocation site of an object is almost always used as part of its label, often combined with additional information, such as the state of the call stack when the object was created. TAJS, WALA, Gatekeeper, JSRefactor, and JSAI all reason about the heap in this way.

In this chapter, we shall give an overview of techniques that are relevant for analyzing JavaScript in a flow-insensitive, subset-based setting. We shall occasionally highlight differences between the various implementations, particularly in places where flow-sensitive analyses have a distinct advantage or disadvantage. Although we focus mainly on subset-based analysis, many of the concepts we discuss are not specific to this type of analysis, but are relevant nonetheless.

After establishing some preliminaries in Section 2.1, we define the core Andersen-style analysis in Section 2.2 and build onto this in Section 2.3 to handle the mechanics of JavaScript as well as some commonly used code patterns. We then conclude in Section 2.4.

2.1 Constraint Systems

We shall specify our static analyses using systems of monotone constraints. They are typeset as inference rules, for example, the following rule specifies that baz must contain all elements present in both foo and bar:

\[ x \in \text{foo}, \ x \in \text{bar} \]
\[ x \in \text{baz} \]

We will use slanted words for variables that constitute the unknowns of the constraint system. The constraint solver will find a value for these such that the system is satisfied. We shall use capital words for inputs to the constraint system; we say such variables are known ahead of time as opposed to the unknowns, which must be computed. We use single-letter symbols \( x, y, z, x, y, z \) etc. for free variables in inference rules; a rule must be satisfied for any instantiation of these variables (i.e. baz must contain all the \( x \)'s that are in both foo and bar). Note that this notation is only used in Chapters 2-3, the publications in Chapters 5-8 introduce their own notation, as originally published.

All unknowns of the constraint system are points in a lattice representing partial information. A lattice we shall use particularly often is the lattice of subsets over some
2. Subset-based Pointer Analysis

finite domain; for instance, the following is the lattice of subsets of \{x, y, z\} ordered by set inclusion:

\[
\begin{align*}
\{x, y, z\} \\
\{x, y\} & \quad \{y, z\} & \quad \{x, z\} \\
\{y\} & \quad \{x\} & \quad \{z\} \\
\emptyset
\end{align*}
\]

We use the convention that the bottom of a lattice is the most precise information, and the top is the least precise. Throughout this dissertation, the only ordering we shall use on subset lattices is set inclusion, as illustrated above, such that the empty set is at the bottom and the entire domain is at the top.

2.2 Andersen-style Analysis

In subset-based points-to analysis, one typically chooses a finite set of abstract objects Obj such that any object at runtime is represented by exactly one abstract object from Obj. If we let \text{VAR} denote the set of variables in the program, and \text{NAME} denote a finite set of property names, the high-level goal is to compute the relations var-pts-to and prty-pts-to on form:

\[
\begin{align*}
\text{var-pts-to} & \subseteq \text{VAR} \times \text{OBJ} \\
\text{prty-pts-to} & \subseteq \text{OBJ} \times \text{NAME} \times \text{OBJ}
\end{align*}
\]

A fact \((v, x) \in \text{var-pts-to}\) denotes that "\(v\) may point to some object represented by \(x\)". A fact \((x, f, y) \in \text{prty-pts-to}\) denotes that "some object represented by \(x\) may point to some object represented by \(y\) in its \(f\) property". In a slight abuse of terminology, the less verbose wordings "\(v\) may point to \(x\)" and "\(x.f\) may point to \(y\)" are generally preferred.

Together, these relations are typically called a points-to graph. Points-to graphs can be visualized as directed graphs with \text{VAR} \cup \text{OBJ} as vertices and \text{prty-pts-to} edges labeled with a \text{NAME}. For example, below is a small program and a corresponding points-to graph soundly approximating the structure of its heap (not showing the \text{Math} object or other native objects):

```java
1 x = { f : {} };
2 y = { f : {} };
3 if (Math.random() > 0.5) {
  4 y = x;
  5 }
6 x.f.g = y.f;
```

We introduce the notation \([x]\) and \([o.f]\) as shorthand for looking up the set of possible points-to targets (called the points-to set) for a variable or property:

\[
\begin{align*}
[x] & := \{ o \in \text{OBJ} \mid (x, o) \in \text{var-pts-to} \} \\
[o.f] & := \{ o' \in \text{OBJ} \mid (o, f, o') \in \text{prty-pts-to} \}
\end{align*}
\]

To introduce the basics of subset-based pointer analysis, let us consider a language with only four basic instructions:
2.2. Andersen-style Analysis

Assign \( x = y \)
Load \( x = y.f \)
Store \( x.f = y \)
Allocate \( x = \text{new} \)

Programs with nested expressions can be brought into this form by moving subexpressions into temporary variables. For instance, the statement on line 6 above can be rewritten as:

\[
\begin{align*}
x.f.g &= y.f \\
\downarrow \\
v1 &= x.f; v2 = y.f; v1.g &= v2;
\end{align*}
\]

In a flow-insensitive analysis, we do not care about the ordering of statements, so we can model the input program as four sets:

\[
\begin{align*}
\text{Assign} &\subseteq \text{Var} \times \text{Var} \\
\text{Load} &\subseteq \text{Var} \times \text{Var} \times \text{Name} \\
\text{Allocate} &\subseteq \text{Var} \times \text{Obj} \\
\text{Store} &\subseteq \text{Var} \times \text{Name} \times \text{Var}
\end{align*}
\]

A load statement \( x = y.f \) is represented by the tuple \( (x, y, f) \in \text{Load} \), but for convenience we use the notation \( \langle\langle x = y.f \rangle\rangle \) instead, and similarly for the other operations:

\[
\begin{align*}
\langle\langle x = y.f \rangle\rangle &= (x, y, f) \in \text{Load} \\
\langle\langle x.f = y \rangle\rangle &= (x, f, y) \in \text{Store} \\
\langle\langle x = y \rangle\rangle &= (x, y) \in \text{Assign} \\
\langle\langle x = \text{new}_o \rangle\rangle &= (x, o) \in \text{Allocate}
\end{align*}
\]

We refer to these instruction sets as the flow-insensitive intermediate representation (IR). Given such an IR, the points-to graph can be computed by solving the following system of constraints:

\[
\begin{align*}
\frac{\langle\langle x = y.f \rangle\rangle, o \in [y]}{[o.f] \subseteq [x]} &\quad \text{[load]} \\
\frac{\langle\langle x.f = y \rangle\rangle, o \in [x]}{[y] \subseteq [o.f]} &\quad \text{[store]} \\
\frac{\langle\langle x = y \rangle\rangle}{[y] \subseteq [x]} &\quad \text{[assign]} \\
\frac{\langle\langle x = \text{new}_o \rangle\rangle}{o \in [x]} &\quad \text{[allocate]}
\end{align*}
\]

It can help to visualize the rules using small points-to graphs, where solid edges represent the conditions for the rule, and dotted edges are the ones being added:

\[
\begin{align*}
\text{[load]} &\quad \begin{array}{c}
\text{y} \rightarrow \\
\text{x} \rightarrow
\end{array} \\
\text{[store]} &\quad \begin{array}{c}
\text{y} \rightarrow \\
\text{x} \rightarrow
\end{array} \\
\text{[assign]} &\quad \begin{array}{c}
\text{y} \rightarrow \\
\text{x} \rightarrow
\end{array} \\
\text{[allocate]} &\quad \\
\end{align*}
\]

We made the simplifying design choice that one abstract object exists per allocation site. This is not the case in general, particularly for context-sensitive analyses that aim to distinguish objects allocated at the same site but in different calling contexts, known as heap cloning or heap specialization. Heap specialization is not specific to JavaScript, so we will stick with this design in order to simplify the presentation.
2.3 JavaScript Mechanics

In the following, we will gradually build onto the core of subset-based analysis to handle JavaScript mechanics. Although we aim to be fairly comprehensive, this is not an exhaustive enumeration of all mechanics, nor is it a complete specification of a points-to analysis. To make things manageable, we will focus on the important mechanics. We will introduce a single mechanic at a time and discuss its impact on the constraint system. More instructions will be added to the IR, but the translation into this IR will not be explained in depth.

There is a fair amount of freedom in the design of points-to analysis; some choices can greatly simplify the constraint system while others leave room for more generality and fine-tuning. We will outline various choices along the way, but ultimately choose some "canonical" design to use as basis when introducing more mechanics. In most cases this canonical design aligns with that of JSRefactor so we can leverage our experiences with this particular implementation.

2.3.1 Coercion to objects

In JavaScript, an expression `x.f` does not require that `x` is an object. If `x` is a primitive value, an object is created in its place by a process called to-object coercion, and `f` is then accessed on this coercion object. In most cases, the coercion object is short-lived and can be garbage collected immediately afterwards. But in general it is possible for a coercion object to escape into a variable and subsequently be manipulated as an ordinary object.

Accurate treatment of to-object coercion requires the pointer analysis to reason about the flow of non-pointer values, such as numbers and strings. A direct approach is to choose a finite set of abstract primitive values `Prim`, and extend the points-to graph to track primitive values:

\[
\text{Value} = \text{Prim} \cup \text{Obj}
\]

\[
\text{var-pts-to} \subseteq \text{Var} \times \text{Value}
\]

\[
\text{prty-pts-to} \subseteq \text{Obj} \times \text{Name} \times \text{Value}
\]

Note that `Prim` and `Obj` need not be disjoint. One can see `Obj` as the set of abstract values that might be objects, and conversely for `Prim`. Coercion to an object can be modeled by a partial function:

\[
\text{ToObj} : \text{Value} \mapsto \text{Obj}
\]

This function maps (abstract) string values to some object label representing the `String` object created by the coercion, and similarly for numbers and booleans. Objects map to themselves and values representing (only) `null` and/or `undefined` map to nil because the coercion raises an exception at runtime in these cases.

We shall work with a simple choice for `Prim` in which there is one abstract value for all strings, one for all numbers, one for each boolean, one for `null`, and one for `undefined`. We also introduce abstract objects `String`, `Number`, and `Boolean` to represent all coercion objects of a given type:

\[
\text{Prim} = \{ \text{str, num, true, false, undef, null} \}
\]

\[
\text{Obj} \supseteq \{ \text{String, Number, Boolean} \}
\]
ToObj\( (v) \) = \begin{cases} 
  \text{String} & \text{if } v = \text{str} \\
  \text{Number} & \text{if } v = \text{num} \\
  \text{Boolean} & \text{if } v = \text{true} \lor v = \text{false} \\
  v & \text{if } v \in \text{OBJ} 
\end{cases}

Given such a coercion function, the \([\text{load}]\) rule can be refined to the following:

\[
\frac{\langle \langle \mathbf{r} = \mathbf{x.f} \rangle \rangle, v \in \llbracket \mathbf{x} \rrbracket, \text{ToObj}(v) = o}{\llbracket o.f \rrbracket \subseteq \llbracket \mathbf{r} \rrbracket} \quad [\text{load}]
\]

To compress the notation a bit, we introduce shorthands for accessing only the object values of a variable, and the values after to-object coercion:

\[
\begin{align*}
\llbracket \mathbf{x} \rrbracket_{\text{IsObj}} & := \llbracket \mathbf{x} \rrbracket \cap \text{OBJ} \\
\llbracket \mathbf{x} \rrbracket_{\text{ToObj}} & := \{ \text{ToObj}(x) \mid x \in \llbracket \mathbf{x} \rrbracket \}
\end{align*}
\]

The \([\text{load}]\) rule can now be simplified to the following:

\[
\frac{\langle \langle \mathbf{r} = \mathbf{x.f} \rangle \rangle, o \in \llbracket \mathbf{x} \rrbracket_{\text{ToObj}}}{\llbracket o.f \rrbracket \subseteq \llbracket \mathbf{r} \rrbracket} \quad [\text{load}]
\]

It is tempting to merge the abstract values for \text{str} and \text{String} (and likewise for numbers and booleans). In this way, the coercion itself becomes an identity except it filters out null and undefined. However, we would lose the ability to distinguish primitives from objects, which turns out to be a major obstacle for accurate treatment of other mechanics, so we will not pursue this approach.

Note that the signature of the \text{ToObj} function precludes the use of an allocation site abstraction for coercions. TAJS uses the allocation site as part of the abstract object produced from a coercion. The same could be done in our setting, but the motivation for doing so in TAJS is to precisely track the primitive values wrapped in a coercion object, which is of little use with our current coarse abstraction of primitive values.

Technically, to-object coercion also occurs during a store operation, although in this case the coerced object is discarded immediately after the store, effectively rendering such stores no-ops. It therefore suffices to handle stores in which the base is already an object:

\[
\frac{\langle \langle \mathbf{x.f} = \mathbf{r} \rangle \rangle, o \in \llbracket \mathbf{x} \rrbracket_{\text{IsObj}}}{\llbracket \mathbf{r} \rrbracket \subseteq \llbracket o.f \rrbracket} \quad [\text{store}]
\]

The only exception is in the presence of property setters, where the coerced object becomes the \text{this} argument. We shall investigate this later, in Section 2.3.14.

### 2.3.2 Absent properties

When reading a property that is not found on the object (or on its prototype chain), \text{undefined} is returned. The previous incarnations of the \([\text{load}]\) rule do not by themselves take this into account. Unfortunately, we can almost never prove the presence of a property while in a purely flow-insensitive setting. User-created objects are empty at creation-time, and without flow information, we must assume that any given load operation could happen before any properties are added to a user-created object. For instance, consider the following two programs:
2. Subset-based Pointer Analysis

\[
\begin{align*}
x &= \text{new} & x &= \text{new} \\
x.f &= 10 & z &= x.f \\
z &= x.f & x.f &= 10
\end{align*}
\]

In the lefthand program, \( z \) obtains the value of the \( f \) property from \( x \), which is 10. The righthand program, \( z \) becomes \text{undefined} because \( f \) is absent at the time. Being flow-insensitive, our analysis cannot distinguish the two programs since one is a permutation of the other, so in both cases, our analysis must assume \( z \) could be \text{undefined}.

We will therefore conservatively assume that any property being read might be absent:

\[
\langle \langle r = x.f \rangle \rangle \quad \text{[load-absent]}
\]

Proving the presence of properties on built-in objects is more feasible in general, since they are not empty initially like user-created objects. We can even imagine ways to prove the presence of properties on user-created objects in some common cases: for instance, we can merge adjacent allocation and store instructions into compound initializers before erasing flow information. For example, \( x = \{ \}; x.f = y \) could be rewritten to a compound operation \( x = \{ f: y \} \). This removes the intermediate program point in which the created object has no \( f \) property. Unfortunately, this interacts poorly with property setters; if a setter for the \( f \) property is defined on \text{Object.prototype}, the two forms are not equivalent, and indeed, the object may not even get its own \( f \) property in this case.

Instead of employing such potentially complicated mechanics, we shall design the analysis so that it can tolerate the imprecision, rather than try and overcome it. If reasoning about the presence of properties is of great importance to the client, it might be better to pursue a flow-sensitive approach, such as TAJS or JSAI.

To our knowledge, none of the flow-insensitive analyses for JavaScript attempt to prove the presence of properties. Gatekeeper does not track the flow of \text{undefined} values at all, hence the [load-absent] rule becomes unnecessary in their case, but as we shall see shortly, the issue interferes with other mechanics.

2.3.3 Prototypes

Every JavaScript object has an internal \textit{prototype link} referring to another object or \texttt{null}. The prototype link is immutable, so it is not possible to create cyclic prototype chains. We represent the possible prototypes of an object by a relation \textit{proto}:

\[
\text{proto} \subseteq \text{Obj} \times \text{Obj}
\]

A fact \((o, o') \in \text{proto}\) indicates that \(o'\) may be the prototype of \(o\). We use the notation \(\text{proto}^*\) to denote its reflexive, transitive closure, and we use the following lookup notation:

\[
\begin{align*}
\text{proto}(o) &::= \{ o' | (o, o') \in \text{proto} \} \\
\text{proto}^*(o) &::= \{ o' | (o, o') \in \text{proto}^* \}
\end{align*}
\]

The load operation traverses the prototype chain and reads the property from the first object where the property is present. This makes it possible for objects to "shadow"
properties from its prototypes. For example, if the below is seen as a concrete heap (no abstraction), then an expression \( x.f \) will result in \( A \) and not \( B \).

\[
\begin{array}{c}
x \quad X \\
\downarrow \quad \text{proto} \\
\downarrow \quad Y \\
\end{array}
\]

However, as discussed previously, a flow-insensitive analysis is ill-equipped to prove the presence of properties. If the above is seen as a points-to graph (i.e. an abstract heap), we must conservatively assume that \( f \) might be absent from \( X \), hence the result of \( x.f \) could be either \( A \) or \( B \) (or \texttt{undefined} if also absent from \( Y \)).

For load operations, we therefore include the values from any object on the prototype chain regardless the order in which they occur. The following rule captures this logic:

\[
\frac{\langle \langle r = x.f \rangle \rangle, o \in [x]_{\text{ToObj}}, o' \in \text{proto}^*(o)}{\langle o'.f \rangle \subseteq [r]}\quad \text{[load]}
\]

Being flow-sensitive, TAJS and JSAI are better equipped to prove the presence of properties, and they can thus reason more precisely about shadowing of properties.

Note that store operations make no use of the prototype chain, except in the presence of property setters, which we shall address in Section 2.3.14.

### 2.3.4 Global variables

Variables in the top-level scope are properties of the \textit{global object}, and can thus be accessed indirectly using load and store operations. We let \( \text{GLOBJ} \in \text{OBJ} \) denote the global object, and use a variable \( \text{glvar} \in \text{VAR} \) as a variable containing it:

\[
\text{GLOBJ} \in [\text{glvar}]_{\text{[glvar]}}
\]

Due to their special nature, global variables are not considered members of \( \text{VAR} \). Instead, all mentions of a global variable are translated into store and load operations on the global object. For example, an expression \( r = x.f \) in the top-level scope would be translated as follows:

\[
\langle \langle v1 = \text{glvar}.x \rangle \rangle \quad \langle \langle v2 = v1.f \rangle \rangle \quad \langle \text{glvar}.r = v2 \rangle
\]

Access to global variables follow the same rules as store and load regarding prototypes, so the translation is faithful in this regard.

### 2.3.5 Dynamic property access

An expression of form \( x[y] \) coerces \( y \) to a string and uses that string as the name being accessed on \( x \). Such expressions are used for array access, since an array entry is just a property whose name is a number string, e.g. '0', '1', '2'. They can also be used to access arbitrary properties, which makes them akin to reflection in a language like Java.
The simplest sound approach is to access every property of the object when a dynamic load or store is encountered. This approach is taken by Gatekeeper:

\[
\langle \langle r = x[y] \rangle \rangle, \ o \in [x]_{ToObj}, \ o' \in proto^*(o), \ n \in NAME \quad \text{[load-dyn]}
\]

\[
\langle \langle x[y] = r \rangle \rangle, \ o \in [x]_{IsObj}, \ n \in NAME \quad \text{[store-dyn]}
\]

The quantifier \(n \in NAME\) can be expensive if \(NAME\) is large. In the case of \([load-dyn]\), we can index \(prty-pts-to\) such that we only enumerate the names for which there is a non-empty points-to set. But \([store-dyn]\) forces us to update a large number of entries in \(prty-pts-to\), which cannot as easily be solved at the data-structure level. An alternative is to introduce a special property name \(unknown \in NAME\), and have dynamic stores write only to this name. We then add an additional rule to ensure that ordinary load operations \(\langle \langle r = x.f \rangle \rangle\) read from \(unknown\) in addition to \(f\).

\[
\langle \langle x[y] = r \rangle \rangle, \ v \in [y]_{ToObj}, \ ToName(v) = n, \ o \in [x]_{ToObj}, \ o' \in proto^*(o) \quad \text{[store-dyn]}
\]

This treatment of dynamic access is sound, but woefully imprecise. When reading a value out of an array, for instance, the built-in functions from the \texttt{Array} prototype, such as \texttt{push} and \texttt{concat}, will flow together with the array entries.

A natural improvement is to narrow down the property names being accessed by an expression \(x[y]\) by considering the values that may flow into \(y\). We use the property name \(num \in NAME\) to represent all numeric properties, and we introduce the function \(ToName\), mapping a value to the name it may correspond to after being coerced to a string:

\[
ToName: \ \text{VALUE} \rightarrow \text{NAME}
\]

\[
\text{ToName}(v) =
\begin{align*}
\text{num} & \quad \text{if } v = \text{num} \\
\text{"undefined"} & \quad \text{if } v = \text{undefined} \\
\text{"null"} & \quad \text{if } v = \text{null} \\
\text{"true"} & \quad \text{if } v = \text{true} \\
\text{"false"} & \quad \text{if } v = \text{false} \\
\text{unknown} & \quad \text{if } v = \text{str} \lor v \in \text{OBJ}
\end{align*}
\]

We return \(unknown\) in cases where we know nothing about the string value after coercion (we will discuss to-string coercion of objects later). We can then configure \([load-dyn]\) and \([store-dyn]\) to use the value of \(y\) to access only the relevant property:

\[
\langle \langle r = x[y] \rangle \rangle, \ v \in [y], \ ToName(v) = n, \ o \in [x]_{ToObj}, \ o' \in proto^*(o) \quad \text{[load-dyn]}
\]

\[
\langle \langle x[y] = r \rangle \rangle, \ v \in [y], \ ToName(v) = n, \ o \in [x]_{IsObj} \quad \text{[store-dyn]}
\]
We need two additional rules for dynamic loads: (1) to ensure that they always read from \texttt{unknown} even if the name is known, as with \texttt{[load-unkn]}, and (2) read from all properties if the name is unknown:

\[
\langle\langle r = x[y] \rangle\rangle, \quad o \in [x]\text{ToObj}, \quad o' \in \text{proto}^*(o) \\
[ o'.\text{unknown} ] \subseteq [r] 
\]

\[
\langle\langle r = x[y] \rangle\rangle, \quad v \in [y], \quad \text{ToName}(v) = \text{unknown}, \quad n \in \text{NAME}, \\
\langle\langle r = x.n \rangle\rangle, \quad o \in [x]\text{ToObj}, \quad o' \in \text{proto}^*(o) \\
[ o'.\text{unknown} ] \subseteq [r] 
\]

It should be noted that the strings "\texttt{NaN}", "\texttt{Infinity}", and "\texttt{-Infinity}" are all number strings. Their representative in \texttt{NAME} is therefore \texttt{num}, and thus, a source-code expression \( r = x.\texttt{NaN} \) should be translated to \( \langle\langle r = x.\text{num} \rangle\rangle \).

One point of criticism is that \texttt{NAME} is a redundant abstraction of string values, since \texttt{unknown} represents any string, hence overlapping with the concrete values represented by other names. Moreover, when \texttt{unknown} is used in \texttt{prty-pts-to}, it is used exclusively for the points-to edges for which the name was unknown to the analysis (which is why regular store operations don’t write to it). Therefore, if one wishes to formalize these abstractions with Galois connections [17], it is best to discard \texttt{unknown} and view it as an implementation-trick to avoid enumeration of \texttt{NAME}.

In TAJS, a slightly different approach is taken, where a field \texttt{default\_other} represents, for any given object \( o \), the set of names not listed in \( o \)'s property map. This abstraction is more amenable to formalization. It is difficult to apply in our setting, however, since it would require the following non-monotone rule, to ensure ordinary load operations can see properties that are only stored dynamically:

\[
\langle\langle r = x.f \rangle\rangle, \quad o \in [x]\text{ToObj}, \quad o' \in \text{proto}^*(o), \quad [ o'.f ] = \emptyset \\
[ o'.\text{default\_other} ] \subseteq [r] 
\]

The apparent monotonicity problem with \([ o'.f ] = \emptyset \) can be resolved by defining a different lattice ordering on \texttt{prty-pts-to}, but such orderings complicate the implementation of constraint solvers. If the non-monotone condition is removed, the rule becomes equivalent to \texttt{[load-unkn]}, effectively bringing us back to the current solution.

More precision can be obtained by using a more precise domain for primitive strings (and numbers, if the ordering of array elements is relevant). TAJS uses constant propagation, which, in some cases, can determine the exact property being accessed. JSAI additionally uses a complement lattice to distinguish special strings, such as \texttt{toString}, which are known to be particularly sensitive to spurious flow. WALA uses a subset-based string domain to track sets of string constants [80].

Madsen and Andreasen [58] presented a range of light-weight string domains and evaluated their ability to prevent spurious flow between dynamic property accesses with computed property names. Their experiments identify a 'champion' domain that can provide a particularly good trade-off in cost and precision; we refer to [58] for the details.

### 2.3.6 Primitive operators

When tracking the flow of primitive values, we must model the effects of operators, such as addition, multiplication, and string concatenation. For this presentation, we are only
interested in the basic type of a primitive value, that is, we wish to know whether it is a string, number, boolean, null, or undefined. This simplifies most operators to the point where the abstract result is a constant; for instance, the result of $x - y$ can only be a number regardless of what values are held in $x$ and $y$.

A notable exception is the plus operator; it doubles as arithmetic addition and string concatenation, so its result can be either a number or a string. Distinguishing numbers from strings is important because the value could be used as the property name in a dynamic property access. An expression $x + y$ yields a number if both operands are numbers, booleans, null or undefined. If either operand is a string, the result is a string. When objects are used as operands, they can be coerced to either a number or a string, depending on the presence of a `valueOf` or `toString` property and whether the object was created by the built-in `Date` constructor. We will model this conservatively so all objects may coerce to either a number or a string.

We introduce the following two operations: `GetPrim` for loading a primitive constant, which suffices to translate most operators, and `Plus`, which can be used to translate the plus operator:

$$\text{GetPrim} \subseteq \text{Var} \times \text{Prim}$$

$$\text{Plus} \subseteq \text{Var} \times \text{Var} \times \text{Var}$$

$$\langle r = v \rangle \quad \text{ ::= } \quad (r, v) \in \text{GetPrim}$$

$$\langle r = x_1 + x_2 \rangle \quad \text{ ::= } \quad (r, x_1, x_2) \in \text{Plus}$$

`GetPrim` simply loads a primitive into a variable:

$$\frac{\langle r = v \rangle}{v \in [r]} \quad [\text{get-prim}]$$

We define predicates `StringLike` and `NumberLike` to identify values that may be coerced to strings and numbers, respectfully, when used as an operand to the plus operator:

$$\text{StringLike}(v) \quad ::= \quad v \in \{\text{str}\} \cup \text{Obj}$$

$$\text{NumberLike}(v) \quad ::= \quad v \in \{\text{num}, \text{true}, \text{false}, \text{null}, \text{undef}\} \cup \text{Obj}$$

If either operand can be coerced to a string, the result is a string; if both can be coerced to a number, the result may be a number (the two cases are not mutually exclusive in the abstract domain):

$$\frac{\langle r = x_1 + x_2 \rangle, \ i \in \{1, 2\}, \ v \in [x_i], \ \text{StringLike}(v)}{str \in [r]} \quad [\text{concat}]$$

$$\frac{\langle r = x_1 + x_2 \rangle, \ v_1 \in [x_1], \ v_2 \in [x_2], \ \text{NumberLike}(v_1), \ \text{NumberLike}(v_2)}{num \in [r]} \quad [\text{add}]$$

TAJS reasons more precisely about primitive values (e.g. it distinguishes positive and negative integers), hence it uses more comprehensive primitive operators.

Gatekeeper does not track primitive values at all. As a consequence, array accesses are not distinguished from arbitrary reflective access, nor is to-object coercion handled faithfully. In the case of Gatekeeper, it should be noted that a program instrumentation is in place to prevent an attacker from abusing these mechanics.
2.3.7 Logical Operators

A logical expression \( x \&\& y \) evaluates and returns the value of \( x \) if it is a false-like value, otherwise it evaluates and returns the value of \( y \). The expression \( x || y \) evaluates and returns the value of \( x \) if it is a true-like value, otherwise it evaluates and returns the value of \( y \). The optional evaluation of the right-hand operand requires no special attention in a flow-insensitive setting, but the optional data flow from the left operand to the result is significant.

The logical operators \( \&\& \) and \( || \) can be translated into assignments, but this is imprecise for an expression such as \( x.xoo \&\& x.xoo() \), which is intended to optionally invoke a \( x.xoo \) method on \( x \) if it is present. If translated to assignments, the \( x.xoo \) function itself would spuriously flow into the result. Some precision can be gained if a separate instruction is used to filter out false-like or true-like values:

\[
\text{Filter} \subseteq \text{VAR} \times \text{VAR} \times \{ \text{true, false} \}
\]

\[
\langle \langle r = x \text{ if } b \rangle \rangle := (r, x, b) \in \text{FILTER}
\]

\[
\text{TRUELIKE}(v) ::= v \in \{ \text{num, true, str} \} \cup \text{OBJ}
\]

\[
\text{FALSELIKE}(v) ::= v \in \{ \text{num, false, str, null, undef} \}
\]

\[
\begin{align*}
\langle \langle r = x \text{ if true} \rangle \rangle, v \in [x], \text{TRUELIKE}(v) & \quad [\text{filter-true}] \\
\langle \langle r = x \text{ if false} \rangle \rangle, v \in [x], \text{FALSELIKE}(v) & \quad [\text{filter-false}]
\end{align*}
\]

The logical operators can then be translated into IR as follows:

\[
\begin{align*}
r & = x \&\& y \Rightarrow \langle \langle r = x \text{ if true} \rangle \rangle \langle \langle r = y \rangle \rangle \\
r & = x || y \Rightarrow \langle \langle r = x \text{ if false} \rangle \rangle \langle \langle r = y \rangle \rangle
\end{align*}
\]

2.3.8 Basic function calls

Functions are first-class values in JavaScript, hence it is undecidable which functions might be invoked at a given call site. Heuristics used in other languages for weeding out potential call targets are generally ineffective for JavaScript. CHA [19] and CGC [4] produce conservative call graphs based on the class hierarchy, but are only applicable to class-based languages. For some dynamic languages, the name and/or arity of the called function can be deduced simply by looking at the call expression, but this is not the case in JavaScript. Although an expression \( x.xoo() \) would appear to call a function named \( xoo \), it can also invoke an anonymous or differently named function. Likewise, the number of arguments need not match the number of parameters declared in the called function.

Most analyses therefore opt to discover the call graph on-the-fly in conjunction with the rest of the analysis. The points-to graph is already equipped to report the objects that might be invoked as a function at any given call site; so we just need to link abstract objects to their corresponding function definition (if any).

Let \( \text{Func} \) be the set of function definitions in the program, and let \( \text{Fundef}(v) \) denote the function definition corresponding to an abstract object \( v \) or \( \text{nil} \) for non-function values. As we are using the allocation-site abstraction, an abstract object cannot correspond to more than one function definition, and this function definition (if any) is known ahead
2. Subset-based Pointer Analysis

of time. We introduce the functions \texttt{params} and \texttt{return} for providing access to the
parameters and returned variable of each function:

\begin{itemize}
    \item \texttt{UserFunc} : function defined by the input program
    \item \texttt{NativeFunc} : native functions
    \item \texttt{Func} = \texttt{UserFunc} \cup \texttt{NativeFunc}
    \item \texttt{fundef} : \texttt{Value} \rightarrow \texttt{Func}
    \item \texttt{params} : \texttt{UserFunc} \rightarrow \texttt{Var}^{*}
    \item \texttt{this} : \texttt{UserFunc} \rightarrow \texttt{Var}
    \item \texttt{return} : \texttt{UserFunc} \rightarrow \texttt{Var}
\end{itemize}

We introduce the following instruction for function calls:

\[ \text{Call} \subseteq \text{Var} \times \text{Var} \times \text{Var}^{*} \]

\[ \langle r = x(\vec{y}) \rangle ::= (r, x, \vec{y}) \in \text{Call} \]

We introduce \texttt{Peripheral} as a datatype representing the input and output points of
functions, and a relation \texttt{call-val} representing the values that may flow to these points:

\[ \text{Peripheral} ::= \text{this} \mid \text{arg}(N) \mid \text{return} \]

\[ \text{call-val} \subseteq \text{Func} \times \text{Peripheral} \times \text{Value} \]

For termination reasons, it should be noted that we only use \texttt{arg}(i) for \( i \) less than the
greatest arity of a function or call expression, so \texttt{Peripheral} can be seen as a finite set.

As usual, we introduce shorthand notation for this new relation:

\[ \left[ f : p \right] ::= \{ v \mid (f, p, v) \in \text{call-val} \} \]

The following rules connect parameters and returns to \texttt{call-val}:

\quad \frac{f \in \texttt{UserFunc}, \ \texttt{params}(f) = \vec{x}, \ i \in \{1 \ldots |\vec{x}|\}}{\left[ f : \texttt{arg}(i) \right] \subseteq \left[ \vec{x}_i \right]} \quad \text{[parm]}

\quad \frac{f \in \texttt{UserFunc}}{\left[ \texttt{return}(f) \right] \subseteq \left[ f : \texttt{return} \right], \ \left[ f : \texttt{this} \right] \subseteq \left[ \texttt{this}(f) \right]} \quad \text{[this-ret]}

The following rules then propagate values between calls and \texttt{call-val}:

\quad \frac{\langle r = x(\vec{y}) \rangle, \ v \in \left[ \vec{x} \right], \ \texttt{fundef}(v) = f, \ i \in \{1 \ldots |\vec{y}|\}}{\left[ y_i \right] \subseteq \left[ f : \texttt{arg}(i) \right]} \quad \text{[call-arg]}

\quad \frac{\langle r = x(\vec{y}) \rangle, \ v \in \left[ \vec{x} \right], \ \texttt{fundef}(v) = f}{\left[ f : \texttt{return} \right] \subseteq \left[ r \right]} \quad \text{[call-ret]}

The propagation of \texttt{this} arguments will be treated in the next section. The value \texttt{undefined} should be propagated into excess function parameters, but for brevity we
omit that from this presentation.
2.3. Method calls and this

A call expression of form \( x.f(\ldots) \) evaluates \( x.f \) as a load, then invokes the resulting function with the value of \( x \) as \( \text{this} \) argument. These are often called \( \text{method calls} \) due to their apparent similarity to method calls in class-based languages. We could model method calls by introducing a \( \text{this} \) parameter to \( \text{CALL} \). That is, a call \( x.f() \) could be translated to \( v_1 = x.f; v_1(x) \) using the convention that the first argument to a function is always \( \text{this} \). Gatekeeper uses that approach, but it is possible to gain precision if modeled more directly. We introduce a separate \( \text{CALLMETHOD} \) instruction for this type of invocation:

\[
\text{CALLMETHOD} \subseteq \text{VAR} \times \text{VAR} \times \text{NAME} \times \text{VAR}^* \]

\[
\langle \langle r = x.f(\vec{y}) \rangle \rangle, o \in [x]_{\text{ToObj}}, o' \in \text{proto}^*(o), v \in [o'.f], \text{FUNDEF}(v) = f, o \in [f : \text{this}] \rangle \text{[method-this]}
\]

Notice that the \( o \) meta-variable is used multiple times in the above rule; both to look up the \( f \) property, and to be passed as \( \text{this} \) argument. To demonstrate how this increases precision compared to Gatekeeper’s approach, consider a call \( x.f() \) where \( x \) may point to two different objects \( A \) and \( B \), with different methods in their \( f \) property:

\[
\text{In this case, the } \text{this} \text{ argument to } \lambda_1 \text{ should be } A, \text{ and the } \text{this} \text{ argument to } \lambda_2 \text{ should be } B, \text{ as shown to the left below. The rule above handles this precisely. If the method call was rewritten as } v_1 = x.f; v_1(x), \text{ then } v_1 \text{ could point to either } \lambda_1 \text{ or } \lambda_2 \text{ while } x \text{ could be either } A \text{ or } B, \text{ and without further information the analysis would be forced to also add the spurious edges shown in the righthand diagram:}
\]

\[
\text{Nothing so subtle happens when propagating argument and return values; the rules for these are essentially a fusion of the load and call rules:}
\]

\[
\langle \langle r = x.f(\vec{y}) \rangle \rangle, o \in [x]_{\text{ToObj}}, o' \in \text{proto}^*(o), v \in [o'.f], \text{FUNDEF}(v) = f, i \in \{1\ldots|\vec{y}|\} \rangle \subseteq [f : \text{arg}(i)] \text{[method-arg]}
\]

\[
\langle \langle r = x.f(\vec{y}) \rangle \rangle, o \in [x]_{\text{ToObj}}, o' \in \text{proto}^*(o), v \in [o'.f], \text{FUNDEF}(v) = f \rangle \subseteq [f : \text{return}] \subseteq [r] \text{[method-ret]}
\]

25
Finally, non-method calls pass the global object as \texttt{this}:

\[ \langle r = x(\vec{y}) \rangle, v \in [x], \text{Fundef}(v) = f \quad \text{GLOBE} \in \llbracket f : \text{this} \rrbracket \]

An expression of form \( x[y](\ldots) \) also behaves as a method call, passing \( x \) as the \texttt{this} argument. We will omit the rules for dynamic method calls since they are rather verbose, but they should be easy to reconstruct given the rules for method calls and dynamic loads.

TAJS uses a similar trick to gain precision. This is also similar to how analyses for Java will filter potential \texttt{this} arguments to match the class containing the invoked method \[71\].

### 2.3.10 Constructor calls

A constructor call \texttt{new x(\ldots)} creates a new object \( o \) with \( x.prototype \) as its prototype object and invokes \( x \) with \( o \) as \texttt{this} argument. If \( x \) returns an object, this is the result of the call expression, otherwise, as is far more common, \( o \) is the result.

As usual we introduce a new instruction:

\[ \text{CallNew} \subseteq \text{Var} \times \text{Obj} \times \text{Var} \times \text{Var}^* \]

\[ \langle r = \text{new}_{o} x(\vec{y}) \rangle \implies (r, o, x, \vec{y}) \in \text{CallNew} \]

The \texttt{Obj} parameter associated with a CallNew represents the object being created. The following rule handles the special mechanics of constructor calls:

\[ \langle r = \text{new}_{o} x(\vec{y}) \rangle, o_f \in [x]_{\text{InObj}}, \text{Fundef}(o_f) = f, \quad \text{proto}(o), o \in \llbracket f : \text{this} \rrbracket, o \in [x] \quad \text{[new-this]} \]

The following two rules are just propagation of argument and return values:

\[ \langle r = \text{new}_{o} x(\vec{y}) \rangle, o_f \in [x]_{\text{InObj}}, \text{Fundef}(o_f) = f, i \in \{1 \ldots |\vec{y}|\} \quad \text{[new-arg]} \]

\[ \langle r = \text{new}_{o} x(\vec{y}) \rangle, o_f \in [x]_{\text{InObj}}, \text{Fundef}(o_f) = f, v \in \llbracket f : \text{return} \rrbracket, v \in \text{Obj} \quad \text{[new-return]} \]

### 2.3.11 Arguments array

Every function implicitly declares a variable \texttt{arguments} referring to an array-like object containing the arguments passed to the function. Inside a function \texttt{function(f,g) \{\ldots\}}, the first argument can be addressed either through the variable \( f \), or through an expression \texttt{arguments[0]}. Likewise, \( g \) coincides with \texttt{arguments[1]}. If more arguments are passed, the additional arguments can also be found in the \texttt{arguments} array.

The following is the approach taken in JSReactor. For each function \( f \) we introduce an abstract object \( \text{ARGOBJ}(f) \) representing \texttt{arguments} arrays created when invoking \( f \):

\[ \text{ARGOBJ} : \text{Func} \to \text{Obj} \]

We require that any argument passed to a function must be contained in the \texttt{num} property of its \texttt{arguments} array:

\[ f \in \text{Func}, i \in \mathbb{N} \]

\[ \llbracket f : \text{arg}(i) \rrbracket \subseteq \llbracket \text{ARGOBJ}(f).\text{num} \rrbracket \]
2.3. JavaScript Mechanics

The quantifier \( i \in \mathbb{N} \) may look difficult at first, but as with \([\text{load-dyn-any}]\), the constraint solver can index its data structure so it only enumerates the \( i \)'s for which a non-empty points-to set flows into \([f : \text{arg}(i)]\).

Let \( \text{argvar}(f) \) denote the arguments variable in function \( f \); the following rule then ensures that the arguments array is made available in this variable:

\[
\text{argvar} : \text{Func} \rightarrow \text{Var} \\
\begin{array}{c}
f \in \text{Func} \\
\text{argobj}(f) \in [\text{argvar}(f)]
\end{array}
\]

If the arguments array is modified, the corresponding parameter variables are also updated, and vice versa, as demonstrated by the following code:

```javascript
function foo(x) {
    console.log(x); // prints original value of x
    arguments[0] = 5;
    console.log(x); // prints 5
    x = 25;
    console.log(arguments[0]); // prints 25
}
```

JSRefactor does not handle this situation, though it can be remedied by adding the following two rules:

\[
\langle\langle x[y] = r \rangle\rangle, \; v \in [y], \; \text{ToName}(v) \in \{\text{unknown, num}\} \\
\text{ARGOBJ}(f) \in [x], \; \text{PARAMS}(f) = \tilde{z}, \; i \in \{1 \ldots |\tilde{z}|\} \\
[r] \subseteq [z_i] \\
\text{PARAMS}(f) = \tilde{z}, \; i \in \{1 \ldots |\tilde{z}|\}, \; \langle\langle z_i = r \rangle\rangle \\
\text{ARGOBJ}(f).\text{num} \subseteq [r]
\]

An alternative approach taken by TSCheck and JSAI, which we shall call the argument vector approach, is to consider the arguments array itself the only argument to a function (in addition this). Parameter references are then translated into loads and stores on the arguments array, similar to how global variables are translated into loads and stores on the global object. This dispenses with the bookkeeping needed to maintain the two-way correspondence between arguments array and parameter variables.

This is not compatible with the use of a single num property for all numeric property names, since all arguments would then flow together. That said, the arguments vector approach is definitely to be recommended for analyses where the abstraction of property names allows for it.

2.3.12 Call and apply

The native function call allows programmers to pass an explicit this argument, circumventing the syntax-oriented mechanic of method calls. A call \( x.\text{call}(y_1, y_2, \ldots) \) invokes \( x \) as a function using \( y_1 \) as this argument, and \( y_{n+1} \) as the \( n \)th argument (i.e. \( y_2 \) is passed as the first argument). Note that \( x.\text{call}(..) \) is not a special syntactic form, call is a property of Function.prototype, which makes it available for use on all function objects. We let call \( \in \text{NativeFunc} \) denote this native function, and use the following rules to model its effect:

\[
v \in [\text{call} : \text{this}], \; \text{FUNDEF}(v) = f \\
[\text{call} : \text{arg}(1)] \subseteq [f : \text{this}], \; [f : \text{return}] \subseteq [\text{call} : \text{return}]
\]
2. Subset-based Pointer Analysis

\[
v \in \llbracket \text{CALL : this} \rrbracket, \ \text{FUNDEF}(v) = f, \ i > 1 \implies \llbracket \text{CALL : arg}(i) \rrbracket \subseteq \llbracket f : \text{arg}(i - 1) \rrbracket
\]  

[native-calls]

All uses of \texttt{call} will have their arguments and return values mixed together, but context-sensitivity will remedy the situation.

The \texttt{apply} function is similar, except all the ordinary arguments are delivered in the form of an array, instead of being passed individually. The call \texttt{x.apply(y, z)} invokes \texttt{x} as a function using \texttt{y} as \texttt{this} argument, with the array entries of \texttt{z} being passed as ordinary arguments. To model it soundly, JSRefactor uses an additional function peripheral \texttt{unknown-arg} to model values passed into an unknown argument, similar to how \texttt{unknown} models values stored in an unknown property name.

<table>
<thead>
<tr>
<th>Peripheral ::= \ldots</th>
<th>unknown-arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{v} \in \llbracket \text{APPLY : this} \rrbracket, \ \text{FUNDEF}(v) = f, \ \texttt{o} \in \llbracket \text{APPLY : arg}(2) \rrbracket \text{isObj}</td>
<td></td>
</tr>
<tr>
<td>\llbracket o.\text{num} \rrbracket \subseteq \llbracket f : \text{unknown-arg} \rrbracket, \ \llbracket o.\text{unknown} \rrbracket \subseteq \llbracket f : \text{unknown-arg} \rrbracket</td>
<td></td>
</tr>
</tbody>
</table>

We will not show the full set of rules for how JSRefactor treats \texttt{apply}, as they are quite verbose, and do not constitute an ideal solution.

The argument vector approach greatly simplifies treatment of the \texttt{apply} function, since one can simply pass \texttt{z} (or a copy of \texttt{z}) as the arguments array, as is done in TSCheck.

### 2.3.13 Implicit calls to toString and valueOf

Several primitive operators, such as \texttt{+} and \texttt{-}, coerce its operands into primitive values. When an object \texttt{o} is coerced to a primitive value, its \texttt{toString} and/or \texttt{valueOf} methods are invoked with \texttt{o} as \texttt{this} argument. This means an expression such as \texttt{x - y} may indirectly cause method calls in which \texttt{x} and \texttt{y} flow into other parts of the program. Instead of modeling these calls explicitly, we shall simply assume that any function stored in a \texttt{toString} or \texttt{valueOf} property may be invoked from anywhere.

\[
o' \in \text{proto}^*(o), \ v \in \llbracket o'.\text{toString} \rrbracket, \ \text{FUNDEF}(v) = f \implies o \in \llbracket f : \text{this} \rrbracket \quad \text{[coerce-string]}
\]

\[
o' \in \text{proto}^*(o), \ v \in \llbracket o'.\text{valueOf} \rrbracket, \ \text{FUNDEF}(v) = f \implies o \in \llbracket f : \text{this} \rrbracket \quad \text{[coerce-num]}
\]

\[
o' \in \text{proto}^*(o), \ v \in \llbracket o'.\text{unknown} \rrbracket, \ \text{FUNDEF}(v) = f \implies o \in \llbracket f : \text{this} \rrbracket \quad \text{[coerce-dyn]}
\]

The analysis used for semi-automatic renaming in Chapter 7 uses a similar to trick to propagate \texttt{this} arguments into methods without looking at call sites.

This trick cannot be used in a flow-sensitive setting, since the state of the object being passed as \texttt{this} depends on where the call is made from. Such implicit calls are especially problematic for systems that use basic blocks to accelerate dataflow analysis, since most primitive operators can then interrupt straight-line control-flow and hence break a basic block. JSAI makes these calls explicit in its IR; this is less problematic in a tree-based IR where there are no basic blocks.

Omitting the flow of \texttt{this} into an implicit call rarely affects soundness outside the relevant \texttt{toString} or \texttt{valueOf} function itself, since values extracted from the \texttt{this} argument typically do not escape from there. The importance of this therefore depends on
whether the client analysis is expected make queries about the values inside the body of a user-defined `toString` or `valueOf` function. In the case of refactoring, this is likely to happen, for instance, because the user might want to rename a property mentioned inside a `toString` method.

### 2.3.14 Getters and setters

In ECMAScript 5, property **getters** and **setters** were introduced to JavaScript. When a function $f$ is stored as the getter of a property $f$ on an object $o$, any attempt to read from the property $o.f$ results in a call to $f$, and the returned value is seen as the value of $o.f$. Conversely, a setter is invoked when attempting to assign to its associated property. For example:

```javascript
var obj = {
  get foo() { return this.bar; },
  bar: 10,
  set baz(value) { console.log(value); }
}
console.log(obj.foo); // prints 10
obj.baz = 20; // prints 20, and does not affect the 'baz' property
console.log(obj.baz); // prints undefined
```

This further complicates loads and stores, as they can now operate as call sites. However, the trick we used for implicit calls to `toString` and `valueOf` can be used again. We first introduce a new relation to represent property getters and setters:

$$
\text{Accessor ::= get | set}
$$

$$
\text{prty-accessor \subseteq OBJ \times ACCESSOR \times NAME \times OBJ}
$$

$$
\begin{align*}
\{ o : \text{get} n \} &\quad ::= \{ o' | (o, get, n, o') \in \text{prty-accessor} \} \\
\{ o : \text{set} n \} &\quad ::= \{ o' | (o, set, n, o') \in \text{prty-accessor} \}
\end{align*}
$$

The set $\{ o : \text{get} n \}$ represents the functions that may be stored as getters on $o.n$, and likewise for setters.

The rules for loads and stores will remain unchanged. Instead, we shall route the return value from a getter into $\text{prty-pts-to}$ where all the relevant load instructions will find it. Likewise, values from $\text{prty-pts-to}$ will be routed into the argument of a setter, so the value from a store will flow to the setter.

$$
\begin{align*}
\text{getter} &\quad o' \in \text{proto}^*(o), \ o_f \in [o' : \text{get} n], \ \text{FUNDEF}(o_f) = f \\
&\quad [f : \text{return}] \subseteq [o.n], \ o \in [f : \text{this}]
\end{align*}
$$

$$
\begin{align*}
\text{setter} &\quad o' \in \text{proto}^*(o), \ o_f \in [o' : \text{set} n], \ \text{FUNDEF}(o_f) = f \\
&\quad [o.n] \subseteq [f : \text{arg}(1)], \ o \in [f : \text{this}]
\end{align*}
$$

The syntax for creating getters and setters, as shown in above example, only permits getters and setters with constant names. Although there is no syntax for dynamically manipulating getters and setters, there is a native function `defineProperty` that allows this, so we must take into account setters whose name is unknown.

$$
\begin{align*}
\text{setter-unkn} &\quad o' \in \text{proto}^*(o), \ o_f \in [o' : \text{set unknown}], \ \text{FUNDEF}(o_f) = f, \ n \in \text{Name} \\
&\quad [o.n] \subseteq [f : \text{arg}(1)], \ o \in [f : \text{this}]
\end{align*}
$$

A similar rule for $\text{getter}$ is not needed, since it needs only write to `unknown`.  

29
The analysis used for semi-automatic rename refactoring in Chapter 7 uses a similar trick to handle getters and setters on object literals.

The situation is again more complicated for flow-sensitive analyses since the values propagated in and out of a getter or setter depend on when the call is made. We are not aware of any flow-sensitive analysis that handles property getters and setters.

2.3.15 With statements

A `with` statement has the form `with(x) {...}`, and creates a scope in which the properties of the `x` object are exposed as variables. There is no syntactic indicator of how a variable inside a `with` statement will resolve. For instance, in the code below, the use of `prefix` inside the `with` can reference the `prefix` variable or the `prefix` property of `obj`.

```javascript
function foo(obj) {
  var prefix = 'Hello␣';
  with (obj) {
    console.log(prefix + msg);
  }
}

foo({msg: 'World'}); // 'Hello World'
foo({prefix: 'Greetings␣', msg: 'Universe'}); // 'Greetings Universe'
```

In JSRefactor, `with` statements are not represented explicitly in the IR. Instead, the translation into IR models them by issuing multiple load, store, and assign instructions for each variable expression inside a `with`. For example, the use of `prefix` inside the `with` will be translated to an assignment and a load instruction, as shown informally here:

```javascript
with (obj) {
  console.log(prefix + msg);
}
```

The instruction `⟨⟨v = prefix⟩⟩` accounts for the possibility that `prefix` could come from the variable declared in the `foo` function (here, we simply assume that `prefix ∈ VAR` refers to the corresponding IR variable). The instruction `⟨⟨v = obj.prefix⟩⟩` accounts for the possibility that `prefix` could be read from `obj`.

Similar translation is performed for `msg` and `console`. These are all the instructions generated for the `with` block:

```javascript
⟨⟨v = prefix⟩⟩
⟨⟨v = obj.prefix⟩⟩
⟨⟨v = glvar.msg⟩⟩
⟨⟨v = glvar.msg⟩⟩
```

The instruction `⟨⟨msg = obj.msg⟩⟩` is spurious because `obj` always has a `msg` property, but the analysis is not equipped to prove the presence of this property, so a specialized instruction would not be able to improve on this. If there was a global variable named `msg`, such a load would cause imprecision.

The `⟨⟨v = obj.console⟩⟩` is also spurious, but there is little consequence to this. The only false flow from such a load comes from the [load-absent] rule. In principle, that rule can safely be disabled for this type of load instruction, but JSRefactor simply accepts the imprecision to reduce the complexity of its instruction set.
With blocks also affect assignments to variables. When assigning to a variable \( x \) inside a \texttt{with} block with argument \texttt{obj}, then if \( x \) is present on \texttt{obj} or one of its prototypes, the assignment will be made to \texttt{obj.x}. To model this in the IR, we introduce a variant of the store instruction that only stores the value if the given property is already present:

\[
\text{StoreIfPresent} \subseteq \text{Var} \times \text{Name} \times \text{Var}
\]

\[
\langle \langle x.f ?= r \rangle \rangle, o \in [x]_{\text{Proto}}, o' \in \text{proto}^*(o), [o.f] \neq \emptyset \rangle \quad \text{[store-present]}
\]

\[
\langle \langle x.f ?= r \rangle \rangle, o \in [x]_{\text{Proto}}, o' \in \text{proto}^*(o), [o'.\text{unknown}] \neq \emptyset \rangle \quad \text{[store-present-dyn]}
\]

Assignments to variables are then translated into these instructions, for example:

```javascript
with (obj) {
  x = 10;
}
⇒ \langle \langle v = \text{num} \rangle \rangle
\langle \langle x = v \rangle \rangle
\langle \langle \text{obj.x} ?= v \rangle \rangle
```

Lastly, a call expression \texttt{f()} inside a \texttt{with} block with argument \texttt{obj} may act as a method call \texttt{obj.f()}, that is, \texttt{obj} is passed as the \texttt{this} argument if \texttt{f} resolved to \texttt{obj.f}. For this case, we issue both regular \texttt{CALL} and \texttt{CALLMETHOD} instructions:

```javascript
with (obj) {
  f();
}
⇒ \langle \langle r = f() \rangle \rangle
\langle \langle r = \text{obj.f()} \rangle \rangle
```

It is possible for a variable expression \( x \) to be nested inside multiple \texttt{with} blocks, and we generate instructions for each block that it might resolve to. We say the syntactic scope of \( x \) is the nearest enclosing scope that declares a variable named \( x \) (ignoring \texttt{with}), or the global scope if no such scope exists. A \texttt{with} block is then considered a possible resolution target for \( x \) if it encloses \( x \) but not its syntactic scope.

Park et al. [64] propose a method for rewriting common uses of the \texttt{with} statement in a semantic-preserving way. SAFE uses this as a preprocessing to eliminate \texttt{with} from the input. This preprocessing could also be used with JSRefactor, although the precision would degrade, as the IR translator would not be able to use the specialized \text{STOREIFPRESENT} instruction (without further analysis). Moreover, such preprocessing can be impractical when the analysis is used for source-code refactoring, since a mapping between the original and rewritten code must be maintained.

TAJS handles \texttt{with} statements differently. They are explicit in the IR and the scope chain is modeled explicitly in the abstract state. TAJS can (sometimes) prove the presence of properties, and thus reason about \texttt{with} blocks shadowing variables in outer scopes, similar to how it reasons about objects shadowing properties from their prototypes.

### 2.3.16 Exceptions

JavaScript has \texttt{throw} and \texttt{try-catch-finally} blocks. Arbitrary values can be thrown by \texttt{throw} and caught using a \texttt{catch}, so a sound points-to analysis must take into account
the flow of values from `throw` to `catch`. Unlike languages like Java, a `catch` catches any exception, regardless of its type at runtime.

The global sink approach is the simplest: we introduce a variable $err \in \text{VAR}$ to contain any value that may have been thrown and caught. Throw statements are translated into assignments to $err$, and catch clauses are translated into assignments from $err$ to the corresponding exception variable:

$$\text{throw } e \Rightarrow \langle \langle \text{err} = e \rangle \rangle$$
$$\text{catch } (e) \{ \ldots \} \Rightarrow \langle \langle e = \text{err} \rangle \rangle$$

We can use one abstract object $\text{ERROBJ} \in \text{OBJ}$ to represent error objects that are created and thrown by the system, e.g. as a result of dereferencing `null` or trying to invoke a non-function value as a function. This is necessary for sound analysis of programs that use `try` but not `throw`.

$$\text{ERROBJ} \in \llbracket \text{err} \rrbracket \llbracket \text{system-error} \rrbracket$$

JSRefactor uses a more precise tracking of exception values, which is similar to how a Java analysis might track exceptions (e.g. [13]), but simpler due to lack of type-based handlers. All functions and call sites have an exceptional result in addition to a normal result. The rules for function calls then propagate exceptional results from callee to callsite, and the IR translator issues assignments from the exceptional result of a call or throw to the catch variable of the enclosing try-catch clause or to the exceptional result of the enclosing function. To be compatible with the trick used for `toString`, `valueOf`, and getters and setters, a global sink is used to route exceptions thrown from such implicitly called functions to all catch clauses.

Exceptional flow is more complicated for flow-sensitive analyses, such as TAJS and JSAI. TAJS uses a factored control-flow graph [14] to maintain an effective basic block structure while taking exceptional flow into account. JSAI has a tree-based IR and encodes exceptional flow using continuations as part of its abstract state.

### 2.3.17 Context-sensitivity

The analysis described so far is context insensitive, that is, it does not distinguish between different calling contexts when analyzing the body of a function. This allows a value to flow into a function from one call-site and be returned at another, as demonstrated by the following minimal example:

```javascript
function id(x) {
    return x;
}
var a2 = id(a1);
var b2 = id(b1);
```

The righthand diagram illustrates the flow of values through `var-pts-to` and `call-val`. Without context-sensitivity, values from `a1` can flow through the body of `id` and into `b2`. This phenomenon can cause drastic loss of precision, which in turn causes loss of efficiency due to the time and space spent propagating spurious flow. This problem is not specific to JavaScript, but for completeness, we will adapt the constraint system to handle context-sensitive analysis.
The clone-based approach aims to avoid spurious flow by (conceptually) analyzing a separate copy of the function for each of the two call-sites, as illustrated:

Instead of sharing the same copy of \( \text{id} \), the two call-sites now have separate copies. Such copies can be introduced for every call-site in the program, and this can be applied recursively, so the call-sites inside a cloned function themselves lead to new copies of other functions. We must obviously restrict ourselves to a finite number of function copies, so some amount of sharing is inevitable for programs that use recursion. Even without recursion, sharing is necessary in practice to avoid an exponential blow-up.

To this end, we introduce a finite set of abstract contexts. Conceptually, we have a copy of each function for every abstract context. We augment the \( \text{var-pts-to} \) and \( \text{call-val} \) relations with an abstract context component to track which calling context (i.e. which function copy) the value belongs to:

\[
\text{Ctx} : \text{abstract contexts}
\]

\[
\text{var-pts-to} \subseteq \text{Ctx} \times \text{Var} \times \text{Value}
\]

\[
\text{call-val} \subseteq \text{Ctx} \times \text{Func} \times \text{Peripheral} \times \text{Value}
\]

In doing so, we require that every function has its own private copy of all IR variables. Later, we will discuss how to handle closure variables in a context-sensitive setting. Rules that do not propagate flow interprocedurally are adapted by introducing a free variable \( \pi \) indicating the context. For example, \([\text{load}]\) rule becomes:

\[
\langle \langle r = x.f \rangle \rangle, o \in [x_\pi]_{T0\text{Obj}}, o' \in \text{proto}^*(o) \rangle [\text{load}]
\]

We use a function \( \text{enter-ctx} \) to decide for any call which copy of the target function to use. In general, \( \text{enter-ctx} \) can be any abstraction of the program state when the call is initiated, although for simplicity we shall restrict it to only depend on the surrounding context and the call-site. We augment call instructions to contain a \( \text{CallSite} \), separating it from other call sites:

\[
\text{enter-ctx} : \text{Ctx} \times \text{CallSite} \rightarrow \text{Ctx}
\]

The rule for propagating arguments and results for a \( \text{Call} \) are then adapted as follows:

\[
\langle \langle r = x(\vec{y}) \rangle \rangle, v \in [x_\pi], \text{fundef}(v) = f, i \in \{1, \ldots, |\vec{y}|\} \rangle [\text{call-arg}]
\]

33
2. Subset-based Pointer Analysis

\[ \langle \langle r = x(\vec{y}) \rangle \rangle, \quad v \in [x_\pi], \quad \text{FUNDEF}(v) = f, \quad \text{ENTER-CTX}(\pi, c) = \pi' \implies [f_{\pi'} : \text{return}] \subseteq [r_{\pi}] \quad \text{[call-ret]} \]

The other rules for function calls are adapted in similar fashion.

The rules [get-prim], [allocate], [glvar], and [system-error] are not straightforward to adapt because their condition does not depend on any IR variables. Consider the following naive adaptation of [get-prim]:

\[ \langle \langle r = v \rangle \rangle, \quad \pi \in \text{Ctx} \]

\[ v \in [r_\pi] \]

This writes to all copies of the IR variable \( r \), which is overly expensive considering that any given function is likely to be unreachable in most contexts. We augment such instructions with their enclosing UserFunc so their rule can be enabled only for the contexts in which the function is reachable. For [get-prim], this amounts to the following changes

\[
\text{GetPrim} = \text{UserFunc} \times \text{Var} \times \text{Prim}
\]

\[ \langle \langle f : r = v \rangle \rangle \quad : \quad (f, r, v) \in \text{GetPrim} \]

\[ \langle \langle f : r = v \rangle \rangle, \quad [f_{\pi} : \text{this}] \neq \emptyset \quad \text{[get-prim]} \]

For this presentation, we exploit the fact that a function is unreachable if it has no possible values for this. Similar adaptations are made to [allocate], [glvar], and [system-error].

Heap specialization can be obtained by separating abstract objects from allocation sites. Let AllocSite be the set of allocation sites (i.e. object literals, constructor calls, function expressions, etc). An abstract object then consists of an allocation site and calling context in which the allocation took place:

\[ \text{Obj} := \text{AllocSite} \times \text{Ctx} \]

The instructions that allocate objects can then be modified to carry an AllocSite instead of an Obj. Although heap specialization is an important parameter in the design of JavaScript analysis, we omit the extended rules from this presentation, as there is little to be learned from looking at them.

2.3.18 Closure variables

Functions can read and write variables declared in their enclosing scopes. For example:

```javascript
function makeGetter(x) {
  function inner(y) {
    return x[y];
  }
  return inner;
}
```

Using a single IR variable for the \( x \) variable in both functions will not facilitate the flow of data from the outer function to the inner. The two functions may be invoked in different abstract contexts, and hence disagree on which entry in \text{var-pts-to} represents their shared copy of the \( x \) variable. In the translation to IR, we can apply closure conversion so environment objects become explicit, as illustrated below:
2.3. JavaScript Mechanics

```javascript
function makeGetter(x) {
function inner(y) {
    return x[y];
}
return inner;
}
```

For this to work, an additional argument must be passed to a function, holding the function object being invoked:

\[
\text{PERIPHERAL ::= funarg | ...}
\]

Without this, the instruction \(\langle venv_1 = \text{inner.env} \rangle\) in the above illustration is not realizable, since there would be no IR variable holding the \text{inner} function inside its own body. Passing the invoked function as an extra argument is straightforward, so we omit the rules for this.

An alternative, taken by TAJS and JSAI, is to keep variables explicit in the IR and model the scope chain as part of the abstract state.

### 2.3.19 For-in loops and complex reflection

A for-in loop `for (var x in y) { ... }` iterates over all the properties of the object in `y`, storing the current property name in `x`. JSRefactor treats `x` as an unknown string:

```javascript
for (var x in y) { ... }  ⇒  \langle x = \text{Str} \rangle
```

It works for programs that only use the for-in loop for dictionary-like objects, i.e. in cases where one might use a for-each loop in Java or C# to iterate over an associative map. However, it lacks the precision to properly handle a commonly used pattern, in which the for-in loop is used to copy all properties from one object to another. For example:

```javascript
function extend(dst, src) {
    for (var x in src) {
        dst[x] = src[x];
    }
}
function StringBuffer() {
    this.array = []
}
extend(StringBuffer.prototype, {
    append: function(x) {
        this.array.push(x)
    },
    toString: function() {
        return this.array.join('')
    }
});
```

In this example, a temporary object with the properties `append` and `toString` is created and passed as the second argument to `extend`. The `extend` function then copies these properties onto the `StringBuffer.prototype` object. When the analysis treats `x` as...
2. Subset-based Pointer Analysis

an unknown string, however, both properties are lumped into the unknown property of `StringBuffer.prototype`, as shown on the following points-to graph:

```
dst → unknown → append → unknown → toString ← append ← toString
(StringBuffer.prototype) → ⟨⟨⟩⟩ (temporary object)
```

When this happens, the imprecision tends to spiral out of control, as arguments and return values flow together in all methods stored on a given object.

It is clear that the statement `dst[x] = src[x]` moves a property from `src` to the same property in `dst`. In practice, this is the case regardless of what `x` is, so in itself the problem would appear to be solvable without doing anything about the for-in loop variable. It is thus tempting to introduce a specialized instruction for property moves:

```
dst[x] = src[x] ⇒ ⟨⟨dst[?] = src[?]⟩⟩
```

This solves the aforementioned problem in cases where the IR translator can detect the pattern. It should be noted that the translation `dst[x] = src[x] ⇒ ⟨⟨dst[?] = src[?]⟩⟩` is not sound in case `x` is an object whose `toString` method returns different values on its first and second invocation. If this is a concern, `x` can be added as an operand to the instruction, so when an object with a user-defined `toString` method flows into `x`, the analysis can fall back on more conservative actions.

Unfortunately, there are more complex incarnations of the `extend` function and different uses of for-in that cannot be captured by the specialized instruction. For example, the `extend` function in jQuery is recursive. It contains a call roughly equivalent to `extend(dst[x], src[x])`, which recursively copies properties from the 'subobjects' of `src` onto the corresponding subobjects of `dst`.

WALA uses correlation tracking, a technique introduced by Sridharan et al. [80] to handle complex reflection, which is particularly common in frameworks, such as jQuery. Using an intra-procedural dataflow analysis, they identify correlated access pairs by the following definition:

“"A dynamic property read \( r \) and a property write \( w \) are said to be correlated if \( w \) writes the value read at \( r \), and both \( w \) and \( r \) must refer to the same property name."" [80]

Using a separate abstract string value for each property name, they demonstrate that context-sensitivity can then be used to solve the problem if blocks containing correlated access pairs are extracted into a function. The `extend` function contains a correlated access pair and would hence the rewritten as:

```
function extend(dst, src) {
    for (var x in src) {
        dst[x] = src[x];
    }
}
```

```
function extend(dst, src) {
    for (var x in src) {
        (function(v) {
            ⟨⟨w = src[v]⟩⟩
            ⟨⟨dst[v] = w⟩⟩
        })(x);
    }
}
```
By using *argument sensitivity* on the parameter to the inner function, the troublesome assignment will get analyzed separately for each property name. By using argument sensitivity for any parameter that is used as the name in a dynamic property access, they generalize the technique to cases where the read and write occur in different functions.

TAJS uses a notion of intra-procedural contexts to achieve a similar effect without introducing new functions \[7\]. Flow-sensitivity complicates the situation, since the iteration order of a for-in loop is specified in ECMA-Script to be implementation-specific, but they demonstrate a way to retain precision in spite of this.

TSCheck capitalizes on the observation that the most complex reflection typically happens during initialization of a library. Instead of analyzing for-in loops in sophisticated ways, it *executes* the top-level code of a given library to obtain a *snapshot* of its concrete state after initialization. The static analysis then proceeds from this state to determine how the library might respond to calls from the application code (see Chapter 8). Dynamic determinacy analysis by Schäfer et al. \[75\] also depends on concrete execution to raise precision, although they take a more conservative approach, and only use information that provably holds true in all executions.

### 2.3.20 Native API and DOM

A sound analysis must take into account the effects of a call to a native function. We previously discussed the built-in `call` and `apply` functions, which we supported by means of constraints specialized to these functions. This trend can be extended to the rest of the API at the cost of significant engineering effort, an approach taken by TAJS \[59\].

For most of the API, a pragmatic approach is to write *stubs*: simplified implementations of the API, written specifically to support the abstractions used by the analysis. For example, the following stub might be used to model the native `Array.prototype.sort` method for an analysis which does not track the ordering of array elements:

```javascript
Array.prototype.sort = function(comparator) {
    var elm = this[0];
    comparator(elm, elm);
}
```

As long as the ordering of elements is irrelevant the primary effect of the `sort` function can be ignored, but the fact that the callback function will be invoked with arbitrary pairs of elements must still be taken into account.

In some cases, an effective 'fallback' strategy can be designed to handle native functions without modeling each native function explicitly. For example:

- JSRefactor models all DOM objects with a single abstract object and uses conservative assumptions about callbacks when a DOM object is invoked as a function (see Section 5.5). A rationale for this is that most refactorings likely pertain to application logic (e.g. it is not possible to rename a property of a DOM object).

- For code completion, Madsen et al. \[59\] devised a way to compensate for missing API modeling by detecting apparent missing flow to certain variables and guessing the shape of the object that might flow there.

- For semi-automatic renaming (Ch. \[7\]), it is sound ignore the native API (although more precision could be gained by modeling it).
2. Subset-based Pointer Analysis

- TSCheck (Ch. 8) inherits the type-based model from TypeScript. In this way, it can analyze code that uses the native API without engineering a separate model for it.

- For Java analysis, Ali and Lhoták [4] leveraged the type system to reason soundly about application code, independently of extern API or libraries. Although most of the Java standard library is itself written in Java, the analysis can benefit greatly from not having to analyze all of it. Unfortunately, their approach is hard to adapt to JavaScript.

Although modeling native API is rarely seen as a theoretically complex problem, it should not be dismissed as an insignificant or easy problem either. The motivation for Madsen’s work mentioned above was that the relevant API for Windows 8 applications is too large to be modeled in practice. In an unpublished work, our own efforts to analyze Firefox extensions were likewise thwarted by the colossal size of the XPCOM interface[1] which is exposed to such extensions written in JavaScript. An effective fallback strategy could not be found for the client in question, and attempts to write stubs for the relevant parts of the XPCOM interface turned out to be impractical due to the sheer size of it. Although Kashyap and Hardekopf [51] demonstrated that JSAI can analyze Firefox extensions if given stubs for the relevant parts of XPCOM, we are not aware of any work that addresses the issue of actually creating stubs for such a large API.

2.3.21 Branch reachability and overloading

JavaScript does not have first-class overloading, but the feature can be mimicked by manually inspecting the types of the arguments and taking different actions in response. For example, suppose the `lookup` function below is intended to be invoked using a number or an object with a number in its index property:

```javascript
var array = []
function lookup(x) {
    if (typeof x.index === 'number') {
        return array[x.index]
    } else {
        return array[x]
    }
}
```

If an object is passed to `lookup`, the analysis will still consider the dynamic access `array[x]` to be reachable, and thus return the values of all properties in `array`, instead of just its numeric properties (recall that objects coerce to any string).

In principle, a flow-insensitive analyses can be upgraded to consider the satisfiability of branching conditions, and only consider instructions that are in reachable branches. For instance, if the analysis could prove that the condition in `lookup` is always true when invoked in some specific context, it could avoid the above spurious flow by not considering the effects of `array[x]` in that context. In this example, however, the condition can only be proven true if `x` is known to have a property `index`, which, as discussed previously, is not feasible to show in a flow-insensitive setting. The converse might be provable, however.

We are not aware of any implementation that tracks branch reachability in a flow-insensitive setting for JavaScript. This could be because the expected precision gains are deemed insufficient due to code patterns similar the above example. We emphasize it here

2.4 Conclusion

We have given an overview of the techniques used to adapt flow-insensitive, subset-based pointer analysis, also known as Andersen-style analysis, to JavaScript. A system of monotone constraints was used to describe the techniques in a concrete way without implementation details.

We have highlighted some pros and cons of flow-sensitivity. Beyond the obvious savings from having only a single abstract heap, the flow-insensitive analysis can greatly simplify implicit call mechanics, while a major drawback being its inability to prove the presence of properties.

We have touched briefly on the topic of cloning-based context-sensitivity, as this is a simple and commonly used technique (albeit not specific to JavaScript). Concrete strategies for choosing contexts were not covered, as we are not aware of any strategies that are substantially better than the well-known $k$-CFA and object sensitivity strategies. Unfortunately, these strategies are often not sufficient (e.g. see [7]) and this motivated our research into alternatives, such as equality-based and field-based analyses.

Subset-based points-to analysis is a powerful and heavily studied foundation for building programming tools, but the feasibility of carrying out such an analysis remains its weak point. When assessing the feasibility of a given analysis problem, it is therefore advisable to look for circumstances in the problem domain that might simplify the analysis, especially with respect to context-sensitivity and the size of the native API. In lieu of such circumstances, advances in the areas of context-sensitivity and partial program analysis would provide the means for more powerful programming tools.
Equality-based pointer analysis treats assignments as bidirectional. That is, an assignment \( x = y \) is treated the same as \( y = x \). In either case, the variables \( x \) and \( y \) are unified, and the analysis will subsequently not be able to distinguish them; any value that may flow into \( x \) is assumed to flow into \( y \), and vice versa. This stands in contrast to the subset-based approach, where it is possible for values to flow in only one direction. The substantial loss in precision is compensated for by a massive improvement in efficiency.

The equality-based analysis computes the aliasing relation explicitly. With bidirectional assignments, aliasing is an equivalence relation and can thus be represented efficiently with a union-find data structure. By contrast, the subset-based analysis reasons indirectly about aliasing, where variables are considered aliased if they point to the same abstract object.

Equality-based analysis was pioneered by Steensgaard \[82\]. Fully context-sensitive versions (up to recursion) have been demonstrated to scale for both Java \[72\] and C programs \[54\]. The technique is difficult to apply to JavaScript, however, and to our knowledge, no work besides our own has found successful applications of the technique in this area.

In this chapter, we will discuss some challenges in adapting equality-based analysis to JavaScript. In order to avoid too much overlap with the previous chapter and the publications in Chapter 7 and 8, this chapter is much less comprehensive.

### 3.1 Steensgaard-style Analysis

As in the previous chapter, we first describe the basics of equality-based analysis on a minimal language with only assignment, store, load, and allocate instructions. We reuse the notation introduced in the previous chapter.

We must reason about aliasing between variables, but also, to some extent, aliasing relations in the heap. For instance, the effects of a store instruction \( x.f = y \) cannot be described using only aliasing between \( x \) and \( y \). We therefore introduce terms that can denote either a variable or the property of a given variable, and compute the aliasing relation \( \equiv \) over these:

\[
\text{TERM} ::= \text{VAR} | \text{VAR} \times \text{Name}
\]

\[
\equiv \subseteq \text{TERM} \times \text{TERM}
\]
We use the notation \( x \odot f \) to denote the compound term \((x, f) \in \text{TERM}\). The effects of a store instruction \( x.f = y \) can be accounted for by the aliasing fact \( x \odot f \equiv y \).

To visualize this as a points-to graph, we use the equivalence classes of \( \equiv \) as vertices, and display an edge from \( x \) to \( x \odot f \) labeled \( f \) for all relevant \( x \) and \( f \). Vertices are labeled with the variables contained in their equivalence class.

For comparison, below is the example from Section 2.2 with its subset-based and equality-based points-to graphs:

\[
\begin{align*}
x &= \{ f : \{ \} \} \\
y &= \{ f : \{ \} \} \\
\text{if } (\text{Math.random()} > 0.5) \{ \\
  &\quad \text{y = x} \\
x.f.g = y.f
\end{align*}
\]

\[
\begin{align*}
x &\rightarrow f \\
y &\rightarrow f \\
g &\downarrow g \\
x, y &\rightarrow f \\
g &\downarrow g
\end{align*}
\]

 Subset-based

 Equality-based

We now introduce the mechanics for computing \( \equiv \). We require that \( \equiv \) is an equivalence relation, so the following rules are part of our constraint system:

\[
\begin{align*}
t_1 \equiv t_1 \quad &\text{[refl]} \\
t_1 \equiv t_2, t_2 \equiv t_1 \quad &\text{[sym]} \\
t_1 \equiv t_2, t_2 \equiv t_3, t_1 \equiv t_3 \quad &\text{[trans]}
\end{align*}
\]

By representing \( \equiv \) with a union-find data structure, these rules are automatically satisfied.

When variables \( x \) and \( y \) are aliased, a value stored on \( x.f \) can be read back through \( y.f \). To account for this, we require that when variables are aliased, their properties are also aliased:

\[
x \equiv y \\
x \odot f \equiv y \odot f \quad \text{[prty]}
\]

Finally, the effects of individual IR instructions are accounted for by the following rules:

\[
\begin{align*}
\langle \langle x = y \rangle \rangle &\quad \text{[assign]} \\
\langle \langle r = x.f \rangle \rangle &\quad \text{[load]} \\
\langle \langle x.f = x \rangle \rangle &\quad \text{[store]}
\end{align*}
\]

There is no rule for the allocate instruction since it cannot introduce aliasing. We will now demonstrate how these rules handle the example shown above. If we ignore allocate instructions, the program has the following IR:

\[
\begin{align*}
\langle \langle y = x \rangle \rangle &\quad \langle \langle v_1 = x.f \rangle \rangle &\quad \langle \langle v_2 = y.f \rangle \rangle &\quad \langle \langle v_1.g = v_2 \rangle \rangle
\end{align*}
\]

The following graph shows how the different terms are unified after only considering the IR-specific rules [assign], [load] and [store]:

\[
\begin{align*}
x &\rightarrow x \odot f \quad \text{[load]} &\quad v_1 &\quad v_1 \odot g \\
\text{[assign]} &\quad y \quad y \odot f \quad \text{[load]} &\quad v_2 &\quad v_2 \odot g \\
\text{[store]}
\end{align*}
\]

The [prty] rule then unifies \( x \odot f \) and \( y \odot f \). By transitivity, \( v_1 \) and \( v_2 \) are then unified, hence the [prty] rule applies again to unify \( v_1 \odot g \) and \( v_2 \odot g \):
By collapsing the connected components of this graph, the points-to graph from the example is obtained. We sometimes refer to the connected components of $\equiv$ (i.e. its equivalence classes) as abstract objects.

So far we reused the IR from the previous chapter in order illustrate the difference in the two approaches. But the use of such an IR is unwarranted with this style of analysis; indeed, whenever the IR translator would issue one of the above instructions, it might as well issue a single "unify" command on the union-find data structure since the effects of the instruction are then completely accounted for. The set $\textsf{VAR}$ would then contain program variables and program expressions in lieu of the temporary variables introduced by the translation into IR. It is convenient for implementation-purposes to forego the IR entirely and perform the analysis from the AST, especially for languages such as JavaScript where designing a useful IR is a complicated endeavor in itself. In Chapters 7-8 we describe an equality-based static analysis with respect to the AST.

### 3.2 Union-Find

The union-find data structure represents equivalence classes over some domain, in our case TERM. The standard data structure can efficiently merge two classes, and find a representative for the class containing a given element. This data structure can be augmented to track the relationship between atomic terms $x$, and their compound terms $x \circ f$ by augmenting all root nodes in the forest with a map from $\textsf{NAME}$ to other nodes. An algorithm for unifying nodes in the augmented data structure is shown in Chapter 7.

An important characteristic of this representation is that nodes are created lazily. A compound term $x \circ f$ belongs to a singleton class (i.e. it has no aliases) if a node does not exist for it. This saves us from actually enumerating all of TERM. We therefore say that a term $t$ 'exists' if it has a node in the data structure, or equivalently, if it has been aliased with a variable (or is itself a variable).

It can sometimes be convenient to address this representation directly when discussing certain language mechanics, so we will freely switch between notions of the data structure and the theoretical framework of equivalence relations and constraint systems.

### 3.3 JavaScript Mechanics

We will discuss a selection of JavaScript mechanics for which the subset-based techniques cannot be carried over directly.

#### 3.3.1 Prototypes

If $x$ is a prototype of $y$, then a value stored on $x.f$ can be read back from $y.f$. This makes prototyping similar to aliasing. A sound way to solve this is by unifying $x$ and $y$ if $x$ might be a prototype of $y$, but this would cause almost all objects to be unified with Object.prototype, and by transitivity then be unified with each other, thus destroying precision completely.
Another approach is to allow any variable to have up to one prototype, by introducing a term \( x \odot \text{proto} \) for each variable \( x \). The following rule can then be added to the system of constraints:

\[
\frac{x \odot \text{proto} \equiv y}{x \odot f \equiv y \odot f} \quad [\text{proto}]
\]

This is little better than the previous suggestion, since for any property \( f \), the \( f \) property of any two objects would be unified. The situation is illustrated by the points-to graph below, where two objects \( A \) and \( B \) have their \( f \) properties unified by the above rule, despite having nothing in common besides inheriting from \texttt{Object.prototype}:

For properties that are actually present on \texttt{Object.prototype}, such as \texttt{hasOwnProperty}, this can be tolerated, but most property names are not present there, and unifying these is unacceptable. We can restrict the rule to only apply for properties that appear to be present on the prototype object:

\[
\frac{x \odot \text{proto} \equiv y, \ \text{exists}(y \odot f)}{x \odot f \equiv y \odot f} \quad [\text{proto}]
\]

The existence check avoids the most rampant imprecision, but a typical pattern can still cause imprecision. Consider the code fragment below, which emulates a class \texttt{Bar} inheriting from a class \texttt{Foo}:

```javascript
function Foo() {}  
function Bar() {}  
Bar.prototype = new Foo();
```

Initially, the \texttt{prototype} property of a function refers to an object whose prototype is \texttt{Object.prototype}. This causes \texttt{Foo.prototype} to be unified with \texttt{Object.prototype}, as illustrated below:

This can be remedied by allowing multiple potential prototypes per abstract object:

\[
\text{proto} : \text{VAR} \xrightarrow{\equiv} \mathcal{P} (\text{VAR})
\]

We use the notation \( \equiv \) to denote a function that is congruent with \( \equiv \), that is, whenever \( x \equiv y \), then \( \text{proto}(x) = \text{proto}(y) \). In implementation-terms, this means we maintain a set
of prototype pointers for each root node in the union-find data structure. The \([\text{proto}]\) rule can be updated as follows:

\[
\frac{y \in \text{proto}(x), \text{exists}(y \circ f)}{x \circ f \equiv y \circ f} [\text{proto}]
\]

TSCheck uses this solution. This significantly reduces the likelihood of an object getting unified with \texttt{Object.prototype}. An alternative might be to allow a single prototype pointer per abstract object \textit{in addition} to \texttt{Object.prototype}, since that object is at the end of almost every prototype chain.

### 3.3.2 Coercion to objects

As with subset-based analysis, we can track the potential primitive values for each abstract object:

\[
\text{primitives} : \text{VAR} \rightarrow \mathcal{P}(\text{PRIM})
\]

If we let \texttt{String.prototype} \in \text{VAR} denote a variable holding the built-in \texttt{String} prototype object, we can model coercion to string objects as follows:

\[
\frac{\text{str} \in \text{primitives}(v), \text{exists}(\text{String.prototype} \circ f), \text{exists}(v \circ f)}{v \circ f \equiv \text{String.prototype} \circ f}
\]

The first existence check serves the same purpose as in the \([\text{proto}]\) rule. The second check ensures we only propagate properties that are needed for the given variable (i.e. the program mentions the \(f\) property of \(v\) or an alias of \(v\)). The second check cannot be used in the \([\text{proto}]\) rule since it would prevent transitive inheritance along the prototype chain.

TSCheck uses a similar technique, except it is grafted together with its propagation of TypeScript types.

### 3.3.3 Dynamic property access

The primitive values of an abstract object can be used to restrict the properties being accessed at a given dynamic property access, but when the name of the property is unknown, we are forced to unify all properties of the given object. For every root node in the union-find data structure, we can keep a flag indicating if it should be treated as an ordinary object or if all its properties should be merged. In the latter case, we only need a single outgoing pointer instead of one for every property, and we say the object is a \textit{dictionary object}.

TSCheck keeps track of individual numeric properties and for this reason there is a an intermediate stage for \textit{array objects} at which only numeric properties are merged, but non-numeric properties remain separated.

For this to be sound, the properties must also be merged with those from the prototype objects. Although the prototype object itself need not be marked as a dictionary object, this can still be highly detrimental to precision, since all properties of \texttt{Object.prototype} will be merged, and this subsequently causes all dictionary objects to have their properties
merged. To illustrate, suppose $A$ and $B$ below have been marked as dictionary objects, and $f_1$ and $f_2$ denote properties that are present on the prototype object:

We have not found a good solution to this. TSCheck unsoundly assumes that the prototype chain is not accessed by dynamic property accesses for which it cannot determine the name.

As with subset-based analysis, we cannot prove the presence of properties, so the primitive value `undefined` is likely to flow anywhere. If `undefined` flows into the property name of a dynamic access, it will access the property named "undefined". Although the same thing happens in subset-based analysis, the consequence is greater for an equality-based analysis where all properties accessed dynamically on a given object will then be merged, even in cases where the name of the property was known to be a single string or `undefined`.

For this reason, TSCheck also assumes that an `undefined` value resulting from an absent property read never flows into a dynamic property name.

### 3.3.4 Logical operators

An expression such as `x.foo && x.foo()` can evaluate either to `x.foo` or the result of `x.foo()`, but the former is only possible if `x.foo` is a false-like value, such as `undefined` or `null`. It is undesirable to unify the two values by allowing both to flow into the result.

TSCheck simply unifies the result with the right operand of the `&&` operator, ignoring the flow of false-like values from the left operand. For the `||` operator, both operands and the result are all unified.

### 3.3.5 Getters and setters

Getters and setters can be partially handled the same way as in the subset-based analysis. If a function $g$ is stored as a getter for property $f$ on some abstract object $x$, we can unify $x$ with the value of `this` in $g$, and unify the return-value of $g$ with $x \circ f$, and similarly for setters.

Getters and setters can be inherited on the prototype chain, however, and in this case the `this` argument is not always the object on which the getter/setter is stored. For example:

```javascript
function Foo() {}
Foo.prototype = {
  get x() { return this.y }
}
var foo = new Foo();
foo.y = 10;
console.log(foo.x); // prints '10'
```

To handle this, the `foo` object must be propagated into the `this` argument of the getter. If the getter is analyzed context-sensitively, it is possible to avoid unifying all instances
of Foo, but since TSCheck is mostly context insensitive, it ignores inherited getters and setters.

3.3.6 For-in loops and complex reflection

Recall the `extend` function discussed in the previous chapter:

```javascript
function extend(dst, src) {
    for (var k in src) {
        dst[k] = src[k];
    }
}
```

In principle, the property move instruction `⟨⟨x[?] = y[?]⟩⟩` can be used to support this kind of reflection in an equality-based setting by unifying the two base objects:

```
⟨⟨x[?] = y[?]⟩⟩
```

In practice, which takes care of most uses of complex reflection, so it has not been a pressing issue.

3.3.7 Full context-sensitivity

Analyzing whole JavaScript programs without any context-sensitivity will in most cases yield a useless result where everything appears to be aliased. Cloning-based strategies, such as k-CFA, can be applied, but it is worth investigating more aggressive strategies, since the efficiency of unification may counterbalance the added cost of context-sensitivity.

One technique of interest is the fully context-sensitive (up to recursion), equality-based DSA analysis by Lattner et al. [54] for analyzing C code. DSA takes a bottom-up approach to context-sensitivity: functions are analyzed before their callers, and a summary object is produced for each function. Every function has a separate abstract heap, and when a call is analyzed, the summary object of the target function is copied into the abstract heap at the call site. Arguments are then unified with their corresponding parameters and returned expressions unified with the result of the call expression. There is an obvious potential for exponential blow-up due to repeated cloning, but the use of unification was reported to prevent the blow-up in practice.

This technique works well partly because the majority of calls in C can in practice be resolved statically to a single call target, but this is almost never the case in JavaScript. This makes it hard to produce function summaries that are independent of their calling context.

A potential solution is to iterate the analysis, so the analysis is restarted until a fixed point is reached, where calls are resolved using the call graph produced by the previous iteration. This is less precise than DSA since it forces all calls to be resolved early, even when the target function depends on the calling context (e.g. for functions take a callback parameter). When such calls are analyzed with many call targets, we observed that unification was often not able to contain the exponential blow-up.

The feasibility of full context-sensitivity in this setting thus remains an open question. Attempts to make TSML context-sensitive using this strategy did not yield a scalable analysis.
The success of TSCheck relies on the use of an initial heap snapshot and the fact that it performs many independent context-insensitive analyses, instead of a single whole-program analysis (details will be discussed in Chapter 8).

3.4 Conclusion

Equality-based analysis for JavaScript has received much less attention than subset-based analysis. This is understandable given the many difficult trade-offs between soundness and precision for which no obvious solution exists. It is particularly difficult to guarantee soundness without ruining precision completely.

Nevertheless, we have found successful applications of the technique in semi-automatic rename refactoring (Chapter 7) and checking correctness of TypeScript interfaces (Chapter 8). The technique is therefore worth considering for applications that can tolerate the potential for unsoundness and have only moderate needs for context-sensitivity. The use of a heap snapshot (see Chapter 8) can sometimes remedy the need for context-sensitivity.

Our success with the technique also motivates further research in this type of analysis. Making the analysis (more) sound without sacrificing precision would substantially expand the range of clients for which it can be used. For C programs, equality-based analyses have been shown to benefit greatly from context-sensitivity [28], and it is well-known that JavaScript analyses benefit from context-sensitivity (e.g. [46, 52]), so advances toward fully context-sensitive equality-based analysis of JavaScript programs would greatly improve the usefulness of the technique. This may in turn open the doors to JavaScript programming tools that have so far been out of reach.
Chapter 4

Other Approaches

The closest related work on pointer analysis for JavaScript was discussed throughout the previous chapters. In this chapter we will briefly discuss some alternative approaches that might also serve as foundation for refactoring and other programming tools.

4.1 Pushdown-based Analysis

Common dataflow problems can be modeled as state reachability in a pushdown system, capitalizing on the fact that state reachability is decidable in an automaton with single stack memory. The stack of the pushdown system can then be used to model one aspect of the input program, which would otherwise be abstracted to some finite domain.

CFL reachability [67] is a graph problem that is equivalent to reachability in a pushdown system. We will use pushdown systems as the formalism of choice, although neither is inherently better than the other.

A static analysis that reduces its dataflow problem to reachability in a pushdown system is said to be pushdown-based. Three notable constructions for this system are:

Heap-sensitive dataflow [67, 79]: Control states represent variables in the program and the stack represents field names. Pushdown configurations represent access paths. An access path "x.f_1.f_2..." is represented by the configuration in control state x with the stack "f_1.f_2...". Store and load instructions move values around in a way that obeys the stack discipline of a pushdown system, and can therefore each be represented as a single transition that pushes or pops a field name, respectively. Assignments are represented as transitions that preserve the stack. Dataflow between two access paths is thus reduced to reachability in the pushdown system.

Call and return can be modeled as assignments. First-class functions can be modeled by passing arguments and return values through fields on the function object.

Context-sensitive dataflow [67, 69]: Control states represent variables and the stack represents the calling context. The variable x in calling context "c_1,c_2,..." is thus represented by the control state x and the stack "c_1,c_2,...". Passing of parameters and return of values obey the stack discipline and are thus represented as transitions, assuming a static call graph is known ahead of time. We are not aware of any effective way to handle store, load, or higher-order functions in this model.
4. Other Approaches

**Context-sensitive control-flow** [20]: Control states are (abstracted) program states without the call stack, which is instead represented by the pushdown stack. Instructions other than call/return are transitions that preserve the stack. Call and return instructions are transitions that push or pop the name of the relevant call site.

The two dataflow approaches are especially hard to adapt to JavaScript. Part of this stems from the fact that instructions that take more than two operands cannot be represented as transitions. For example, a dynamic property load takes a base object, a property name, and a result. If modeled as a transition, it can only connect two of these operands (as source and destination control states), the third will necessarily be left out. The control-flow model does not have this limitation, but is more expensive due to its large state space.

Horn and Might [86] devised a sound analysis for JavaScript based on the context-sensitive control-flow construction. Their design and implementation operate on a variant of $\lambda_{JS}$ (see Guha et al. [40]). We are not aware of any evaluation of its precision and scalability, but we suspect that overhead from the translation into a low-level language such as $\lambda_{JS}$ is holding it back, so such an evaluation would likely not represent the true potential of the technique.

Vardoulakis and Shivers presented CFA2 [88], which is also a pushdown-based analysis using a construction similar to the context-sensitive control-flow model. Vardoulakis [87] implemented CFA2 for JavaScript with some pragmatic choices favoring precision and efficiency over soundness. It was demonstrated to be fast and precise for many inputs, and as it computes a subset-based heap abstraction, it could likely be used as the foundation for refactoring, as in Chapter 5. It is naturally more precise and less scalable than the context-insensitive equality-based analyses we use in Chapter 7–8, but exactly where it sits in the spectrum between the context-sensitive subset- and equality-based analyses is unaccounted for.

It remains unclear how much precision we stand to gain from full context-sensitivity when the heap abstraction remains insensitive to the calling context, and we suspect that some innovation in this area is needed. Vardoulakis reported in his dissertation [87] that the imprecise heap is indeed a source of imprecision that can cause CFA2 to timeout.

The heap-sensitive and context-sensitive analyses cannot be combined directly [68]; the combined pushdown system needs two stacks, and then reachability becomes undecidable. If recursive functions are eliminated somehow, one of these stacks become bounded, and the problem becomes decidable again, although at an exponential cost (effectively from inlining all functions). Sridharan et al. [78] used this idea to analyze Java programs with a demand-driven, refinement-based analysis, which was reported to avoid the exponential blow-up in practice. Adapting their technique to JavaScript could provide for a powerful analysis tool, but since it is based on the two dataflow models, it is not obvious how this could be done.

4.2 Dynamic Analysis

Dynamic analysis operates by observing, and possibly influencing, the execution of a program. Like static analysis, this can serve as a foundation for programming tools.

---

1. Their use of a "global heap" makes the relation to pushdown systems a bit more complicated.
4.2. Dynamic Analysis

Neither approach is strictly better than the other: for example, dynamic analysis depends on the presence of a high-coverage test suite, or another source of inputs, while static analysis can be imprecise due to abstraction.

The refactoring browser for Smalltalk by Robert et al. [70] uses dynamic analysis to refactor Smalltalk programs, and a similar technique could be developed for JavaScript. Lack of coverage in the dynamic analysis can be tolerated if resorting to semi-automatic techniques, as in Chapter 7, but in general, the effect of such a tool relies on high coverage. We suspect that non-local refactorings such a property renaming require particularly high code coverage, which is one of the main reasons we favor a static approach in our work. Another reason to prefer static analysis is that it requires no configuration from the end-user: the user does not have to instruct the tool on how to execute the test suite.

TSCheck performs dynamic analysis since it executes the initialization code of the library being analyzed to obtain a snapshot of its state. However, only the initialization code is executed, so no test suite is required, and no configuration burden is imposed on the end-user.

EventRacer by Raychev et al. [66] uses a mostly dynamic analysis to detect event races in web pages. In principle, EventRacer can detect more races with greater coverage, but since event races are particularly common during initialization of a web page, it can detect many races automatically without a deep exploration of the state space.

JSBAF by Wei and Ryder [91] analyzes web pages using blended analysis: a dynamic analysis followed by a static points-to analysis. The dynamic analysis resolves dynamically loaded and generated code, and provides a call graph to be used by the subsequent static analysis. In principle, the resulting points-to graph could be used as the foundation for automatic refactoring, as in Chapter 5, although the refactoring logic is not equipped to refactor dynamically generated code, even if actual code being generated is known.

Dynamic determinacy analysis by Schäfer et al. [75] uses a dynamic analysis to determine facts about a program that are guaranteed to hold in any execution, which, unlike the above approaches, allows for a subsequent sound, static points-to analysis. This information can be leveraged by a static analysis to improve precision and scalability. It was demonstrated to work well in combination with WALA, which can also serve as a foundation for refactoring, as in Chapter 5.

These examples all combine some degree of dynamic and static analysis, although there is a notable difference between the snapshot technique in TSCheck and the others. TSCheck uses concrete execution up to a given point in time (the end of the top-level code), and uses abstract interpretation to reason about what might happen after this point. The others use abstract interpretation along the whole time line, exploring a state space that carries a certain similarity to the observed traces. This is illustrated below, where the gray regions indicate where traces are discovered using abstract interpretation:
4. Other Approaches

One advantage of the snapshot technique is that the static analysis does not have to reason about the potentially complex initialization code at all, whereas the blended analyses must still analyze the code, albeit with some assistance from the dynamically collected information. However, if the client intends to reason about the initialization code, for instance, to detect event races or other suspicious behavior, the snapshot technique does not apply. The blended approach can benefit from dynamically collected information after the snapshot point, and may thus have higher precision because of this. In principle, the two approaches can be combined, although we are not aware of any work that does this for JavaScript.

4.3 Type Systems for JavaScript

The type checking or type inference procedure of a static type system can be seen as a static analysis, although this terminology is used sparingly. Type systems and programming languages are typically designed together as a unit, wherein a delicate balance must be reached between the complexity of types, restrictions imposed upon the language, and the benefits gained from the types. Very different challenges and techniques are faced in this context, hence type systems and static analysis are best thought of as distinct fields that are known to overlap in some areas.

A notable difference between the two worlds is that most type systems will "reject" a program that fails to conform to the rules, and instead of trying to "understand" the offending program fragment, a warning about it will be issued to the user. In contrast, abstract interpreters will typically try to produce some information about the program, regardless of how complex it is.

Several type systems have been proposed for JavaScript. Some require type annotations, such as TeJaS \[56\] and DJS \[15\], and support powerful types to handle the tricky mechanics of JavaScript. Others rely on type inference but only support idealized subsets of the language, such as JS0 by Anderson et al. \[6\]. These type systems can likely support powerful programming tools, but such tools would naturally not apply to plain JavaScript.

TypeScript \[60\] extends the language by adding optional types and syntax for class and module declarations. Unlike most type systems being researched for JavaScript, optional types do not aim to provide any safety guarantees, but can nonetheless provide the basis for programming tools, such as code completion. Its liberal type system seems to be popular in the industry, and it has mechanisms for interfacing with libraries written in plain JavaScript. We refer to Chapter \[8\] for a more detailed look at TypeScript and its relation to the JavaScript ecosystem.
Part II

Publications
Chapter 5

Tool-supported Refactoring for JavaScript

This is chapter is a copy of an OOPSLA’11 article [27] by Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip.

Abstract

Refactoring is a popular technique for improving the structure of existing programs while maintaining their behavior. For statically typed programming languages such as Java, a wide variety of refactorings have been described, and tool support for performing refactorings and ensuring their correctness is widely available in modern IDEs. For the JavaScript programming language, however, existing refactoring tools are less mature and often unable to ensure that program behavior is preserved. Refactoring algorithms that have been developed for statically typed languages are not applicable to JavaScript because of its dynamic nature.

We propose a framework for specifying and implementing JavaScript refactorings based on pointer analysis. We describe novel refactorings motivated by best practice recommendations for JavaScript programming, and demonstrate how they can be described concisely in terms of queries provided by our framework. Experiments performed with a prototype implementation on a suite of existing applications show that our approach is well-suited for developing practical refactoring tools for JavaScript.

5.1 Introduction

Refactoring is the process of improving the structure of software by applying behavior-preserving program transformations [29], and has become an integral part of current software development methodologies [12]. These program transformations, themselves called refactorings, are typically identified by a name, such as RENAME FIELD, and characterized by a set of preconditions under which they are applicable and a set of algorithmic steps for transforming the program’s source code. Checking these preconditions and applying the transformations manually is tedious and error-prone, so interest in automated tool support for refactorings has been growing. Currently, popular IDEs such as Eclipse1 VisualStu-

1 http://www.eclipse.org/
and IntelliJ IDEA provide automated support for many common refactorings on various programming languages. In addition, there is much recent research literature on soundly performing a variety of refactorings (see Section 5.7 for an overview).

However, most research on refactoring has focused on statically typed languages, such as Java, for which expressing the preconditions and source code transformations can take advantage of static type information and name resolution. Refactoring for dynamic languages such as JavaScript is complicated because identifiers are resolved at runtime. Of the few previous approaches to refactoring for dynamically typed languages, the most well-developed one can be found in the Smalltalk Refactoring Browser [70], which relies on a combination of runtime instrumentation and the existence of a test suite to ensure that behavior is preserved. By contrast, we aim for a sound technique that does not require comprehensive test suites.

In this paper, we present a framework for refactoring programs written in JavaScript, a dynamically typed scripting language that has become the lingua franca of web browsers. To understand why implementing even simple refactorings in JavaScript is more challenging than implementing analogous refactorings for statically typed languages such as Java, consider the Rename Field refactoring in Java. A key requirement when renaming field \( f \) of class \( C \) to \( g \) is to identify all references to that field so they can be renamed consistently. Renaming all references to a field is easy for Java programs since static type information is available. For example, an expression of the form \( e.f \) where the static type of \( e \) is \( C \) definitely refers to the renamed field. In contrast, the corresponding task for a Rename Property refactoring in JavaScript is in general impossible to solve exactly by static means. While fields in Java are statically declared within class definitions, properties in JavaScript are only associated with dynamically created objects and are themselves dynamically created upon first write. Further complications arise from other dynamic features of JavaScript, such as renaming, and novel JavaScript-specific refactorings that target desirable programming idioms advocated by influential practitioners [18].

We describe a methodology for implementing automated refactorings on a nearly complete subset of the ECMAScript 5 language [22], the chief omission being dynamically generated code (i.e., `eval`). Our approach relies on static pointer analysis for JavaScript to define a set of general analysis queries. We have used this methodology to implement both well-known traditional refactorings, such as renaming, and novel JavaScript-specific refactorings that target desirable programming idioms advocated by influential practitioners [18].

In the process, we have devised various techniques to handle JavaScript’s highly dynamic features and lack of static typing. For example, while naively over- or under-approximating the set of expressions \( e.f \) that must be modified when a property \( f \) is renamed (e.g., using a conventional must- or may-point-to analysis) would be unsound, we describe an algorithm that over-approximates this set in a safe manner. We also ensure, through preconditions that can be expressed in terms of the analysis queries, that behavior is preserved in the presence of complex JavaScript features such as reflective `for-in` loops, first-class functions, and prototype-based inheritance. In cases where we cannot guarantee behavior preservation, refactorings are prevented from being applied.
5.2. Motivating Examples

We have specified and implemented three refactorings using our approach: RENAME (which is a generalization of the previously mentioned RENAME PROPERTY), ENCAPSULATE PROPERTY, and EXTRACT MODULE. We have evaluated the quality of our implementations by applying these refactorings systematically to a set of 50 benchmark programs, measuring how often refactorings are applied successfully and analyzing causes for rejection. Our results show that most refactorings are performed successfully and rejections are generally justified by a real danger of unsoundness. This demonstrates that our approach is a viable basis for implementing refactoring tools for JavaScript.

In summary, the major contributions of this paper are as follows:

- We present a **framework for specifying and implementing JavaScript refactoring**, based on a set of analysis queries on top of a pointer analysis.
- We give **concise, detailed specifications of JavaScript-specific refactorings** expressed using the framework. To the best of our knowledge, we are the first to give such specifications in the context of JavaScript.
- We **experimentally validate** our approach by exercising a prototype implementation of the framework and the refactorings on a set of JavaScript benchmarks. We demonstrate that the preconditions of our specifications are not overly conservative, and that a relatively simple pointer analysis appears to suffice in practice for many programs ranging in size from 300 to 1700 lines of code.

The remainder of this paper is organized as follows. Section 5.2 introduces a motivating example to illustrate the challenges that arise in defining several refactorings for JavaScript. Section 5.3 presents a framework of analysis queries based on pointer analysis. Section 5.4 shows how the three refactorings under consideration are expressed using this framework. Details of the implementation are described in Section 5.5, while Section 5.6 gives an evaluation of our refactorings on a set of JavaScript benchmarks. Related work is discussed in Section 5.7. Finally, conclusions are presented in Section 5.8.

5.2 Motivating Examples

Figure 5.1 shows a small JavaScript program that we will use to illustrate some of the challenges of refactoring JavaScript programs. Part (a) of the figure shows a library that defines two shapes: circles and rectangles. Part (b) shows a client application that uses this library to draw a number of such shapes of randomly chosen sizes at random coordinates in the browser. We will first explain some key details of this program, and then discuss some of the issues raised by applying the RENAME, ENCAPSULATE PROPERTY, and EXTRACT MODULE refactorings.

5.2.1 A JavaScript Example Program

As a prototype-based language, JavaScript does not have built-in support for classes. Instead, they are commonly simulated using constructor functions. In the example of Figure 5.1, two constructor functions are provided: **Circle** (lines 1–11) and **Rectangle** (lines 13–24). These enable the programmer to create circle and rectangle objects using the **new** operator (e.g., line 41). Constructor functions typically contain statements to
Section 5. Tool-supported Refactoring for JavaScript

(a)

```javascript
function Circle(x, y, r, c) {
    this.x = x;
    this.y = y;
    this.radius = r;
    this.color = c;
    this.drawShape = function (gr) {
        gr.fillCircle(new jsColor(this.color),
                      new jsPoint(this.x, this.y),
                      this.radius);
    }
}
```

```javascript
function Rectangle(x, y, w, h, c) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
    this.color = c;
    this.drawShape = function (gr) {
        gr.fillRectangle(new jsColor(this.color),
                        new jsPoint(this.x, this.y),
                        this.width, this.height);
    }
}
```

```javascript
Rectangle.prototype.getArea = function() {
    return this.width * this.height;
}
```

(b)

```javascript
function r(n) { return Math.round(Math.random() * n); }

function drawAll(sh) {
    var gr = new jsGraphics(document.getElementById("canvas"));
    sh.map( function(s) { s.drawShape(gr); });
}

var shapes = [];
for (var i = 0; i < 500; i++) {
    var o = new jsColor().rgbToHex(r(255),r(255),r(255));
    switch(r(2)){
        case 0:
            shapes[i] = new Circle(r(500),r(500),r(50),o);
            break;
        case 1:  
            shapes[i] = new Rectangle(r(500),r(500),r(50),r(50),o);
            alert(shapes[i].getArea());
            break;
        case 2: 
            break;
    }
}
drawAll(shapes);
```

Figure 5.1: Shapes example. Part (a) shows a small library that defines several types of shapes. Part (b) shows a small client application that uses the library to draw shapes in the browser.
initialize a number of object properties, which are not explicitly declared but created upon the first write. For example, the constructor for Circle creates and initializes properties \( x, y, \) radius, and color (lines 2–5) by assigning them values that are passed in as parameters to the function, and similar for Rectangle.

Both also create properties drawShape on line 6 and line 19 that contain functions to display the appropriate geometric shape. These functions can refer to their receiver object using \( \text{this} \) expressions, and thus act like methods.

Function Rectangle shows another way of emulating methods that makes use of JavaScript’s prototype-based nature. Functions like Circle and Rectangle are themselves objects, and hence can have properties. In particular, every function object has a prototype property that is implicitly initialized to an empty object. On line 25 we create a property getArea in this object by assigning it a function that computes the area of a rectangle. Every object created by invoking \( \text{new Rectangle}(...) \) has an internal prototype property, which references the object stored in Rectangle.prototype. When a property \( x \) is looked up on this object, but the object does not itself define property \( x \), the internal prototype is searched for \( x \) instead.

Thus, every rectangle has both a getArea and a drawShape property, the latter defined in the object itself, the former defined in its internal prototype. But while every rectangle has its own copy of drawShape (created on line 19), there is only one copy of getArea, which is shared by all rectangles.

Function \( r \) (line 28) returns a random value between 0 and its argument \( n \). Function drawAll (lines 30–34) takes as an argument an array shapes, and on line 33 uses a closure and the map function to invoke drawShape on each element of the array. Lines 36–49 contain a sequence of statements that are executed when the page containing the script is loaded. This code creates an array of 500 randomly colored shapes of various kinds, displaying the area of every rectangle upon creation on line 45, and then invokes drawAll to draw these shapes in the browser.

Note that in the invocation \( \text{shapes}[i].\text{getArea()} \) on line 45, the function to be invoked is found on the internal prototype object of \( \text{shapes}[i] \), but its receiver object (i.e., the value of \( \text{this} \)) is \( \text{shapes}[i] \) itself, not the prototype object. This ensures, for instance, that the property access \( \text{this}.\text{width} \) in line 26 refers to the property defined on line 16.

We will now discuss the issues that arise when three refactorings—Rename, Encapsulate Property, and Extract Module—are applied to the example program of Figure 5.1.

5.2.2 Rename

We begin by considering some applications of the Rename refactoring to the example program of Figure 5.1. In JavaScript, there are no property declarations. Although it is natural to think of the assignment to \( \text{this}.x \) in Circle as a declaration, it is just a write access to a property \( x \) that is created on the fly since it does not exist yet. The absence of declarations and static typing information makes refactoring more difficult because it is necessary to determine all property expressions in a program that may refer to the same property and rename them consistently. We consider a few examples:

4 The functions are implemented using jsDraw2D, a graphics library for JavaScript, which is available from http://jsdraw2d.jsfiction.com/
• The property expression `this.x` on line 2 in `Circle` can be renamed to `xCoord`. This requires updating the property expression `this.x` on line 8 to `this.xCoord` as well. However, there is no need to rename the property expression `this.x` on line 14, because the properties accessed on lines 8 and 14 must reside in different objects. If we nevertheless do decide to rename `this.x` on line 14 to `this.xCoord` as well, then the subsequent property expression on line 21 must also be changed to `this.xCoord`.

• Refactoring the property expression `this.drawShape` on line 6 in `Circle` to `this.draw` requires that the property expression `this.drawShape` on line 19 in `Rectangle` is refactored to `this.draw` as well: the receiver `s` in the expression `s.drawShape(gr)` on line 33 can be bound to a `Circle` or a `Rectangle` object, and therefore the methods have to be renamed consistently. Note that `Circle` and `Rectangle` are completely unrelated; in particular there is no prototype relationship.

As these examples illustrate, the key correctness requirement for renaming is name binding preservation—each use of a property in the refactored program should refer to the same property as in the original program. Name binding preservation is also a natural correctness condition for other refactorings, as we describe below. Schäfer et al. [74] used this condition successfully to provide sound automated refactorings for Java, including renaming. Unfortunately, their techniques rely on explicit declarations and static scoping, so they are not directly applicable to JavaScript.

A natural approach is to use a static pointer analysis to approximate name binding information. However, a naive use of pointer analysis would be unsound. For example, consider the renaming of `this.drawShape` on line 6 in `Circle` described above. Renaming only expressions that must point to the same property as the one referenced on line 6 is insufficient. A sound must-point-to analysis could indicate that there is no other access of `drawShape` that must definitely point to the same property, therefore requiring nothing else to be renamed. On the other hand, renaming only expressions that may point to the property referenced on line 6 is also insufficient. For example, a sound may-point-to analysis could exclude the property expression `this.drawShape` on line 19 in `Rectangle` since it definitely accesses a distinct property from that of `Circle`. However, that expression must in fact be renamed to preserve program behavior, as we saw above. We define a notion of relatedness in Section 5.3 based on may-points-to information, which captures the set of property expressions in a program that are affected by renaming a particular expression `a`.

We now consider a minor variation on the client application where a function `dbl` has been added (lines 1–7) as shown in Figure 5.2. The client application is the same as in Figure 5.1(b) except that `dbl` is used to double the radius of circles created in line 23.

The function `dbl` takes an argument `c`, which is assumed to be a `Circle` object, and returns a new `Circle` object at the same coordinates but with a doubled radius. This function illustrates several interesting features of JavaScript. First, on line 2 the constructor of `Circle` is called without any explicit arguments, which causes the special value `undefined` to be passed as default argument. Second, line 3 shows a `for-in` loop, in which the variable `a` iterates through the names of properties in the object pointed to by parameter `c`. Line 4 also provides several examples of a dynamic property expression. For example, the dynamic property expression `c[a]` on that line refers to the property of
5.2. Motivating Examples

```javascript
1 function dble(c) {
2     var nc = new Circle();
3     for (var a in c) {
4         nc[a] = (a != "radius") ? c[a] : c[a]*2;
5     }
6     return nc;
7 }
8
9 function r(n) { return Math.round(Math.random() * n); }
10
11 function drawAll(sh) {
12     var gr =
13     new jsGraphics(document.getElementById("canvas"));
14     sh.map( function(s) { s.drawShape(gr); });
15 }
16
17 var shapes = [];
18 for (var i = 0; i < 500; i++){
19     var o = new jsColor().rgbToHex(r(255),r(255),r(255));
20     switch(r(2)) {
21         case 0:
22             shapes[i] =
23                 dble(new Circle(r(500),r(500),r(50),o));
24             break;
25         case 1:
26             shapes[i] =
27                 new Rectangle(r(500),r(500),r(50),r(50),o);
28             alert(shapes[i].getArea());
29             break;
30     }
31 }
32 drawAll(shapes);
```

Figure 5.2: Modified client application, with dble function added.

c named by the value of a. Together, these reflective features have the effect of copying all property values from the argument c into the corresponding property of the newly created object, except for radius, which is also multiplied by two. These features pose some challenges for refactoring:

- Applying Rename to this.radius on line 4 is problematic because of the for-in loop and dynamic property expression in dble. For example, renaming the property expression to this.rad would require changing the string constant "radius" on line 4 in order to preserve behavior. In general, dynamic property expressions may use values computed at runtime, which would thwart any static analysis. In order to ensure that dynamic property expressions do not cause changes in program behavior when applying the Rename refactoring, our approach (as detailed in Section 5.4) is to conservatively disallow the renaming of any property in any object on which properties may be accessed reflectively. Hence, in this example, we disallow renaming any of the properties in Circle objects.

- The names of the drawShape methods in Circle and Rectangle must be kept consistent, because the call on line 14 may resolve to either one of these, as we explained above. Since we now disallow renaming any of the properties in Circle, we must also disallow renaming drawShape in Rectangle.
• The remaining properties of Rectangle, i.e., x, y, width, and height can still be renamed.

5.2.3 Encapsulate Property

In Java, the ENCAPSULATE FIELD refactoring can be used to encapsulate state by making a field private and redirecting access to that field via newly introduced getter and setter methods [29, page 206]. Unfortunately, JavaScript does not provide language constructs to control the accessibility of properties in objects: If a function has a reference to an object, it can access any property of that object. Such a lack of encapsulation is problematic because it leads to code that is brittle, and hard to understand and maintain.

A commonly used technique, suggested for instance in Crockford’s popular textbook [18], uses local variables of constructor functions to simulate private properties. Local variables in JavaScript (i.e., variables declared using the var keyword) can only be accessed within the scope of the declaring function. In the case of constructor functions, local variables exist as long as the object exists, and they can only be accessed by functions defined within the constructor function itself. The basic idea of the ENCAPSULATE PROPERTY refactoring is to encapsulate state by storing values in local variables instead of properties of objects, and to introduce getter/setter methods to retrieve and modify them.

Figure 5.3 shows how the library of Figure 5.1 is changed by applying the ENCAPSULATE PROPERTY refactoring to the width property of Rectangle, with changed bits of code highlighted in gray. The width property was changed into a local variable on line 17, and methods getWidth and setWidth were introduced on lines 20–25. Furthermore, the property expression this.width was replaced by a call to getWidth on line 33. Note that there was no need to introduce a call to getWidth on line 29 because the width variable can be accessed directly. No calls to setWidth need to be introduced since there are no write accesses to width. The source code of the client application in Figure 5.1(b) is unaffected by this refactoring because it does not access the width property.

Name binding preservation is a key correctness condition also for the ENCAPSULATE PROPERTY refactoring, but there are other issues as well.

Encapsulating the width property of Rectangle did not cause any problems, and all other properties of Rectangle can be encapsulated similarly. However, this is not the case for the properties of Circle. To see this, consider a situation where the radius property of Circle is encapsulated in a scenario where the library is refactored together with the modified client application of Figure 5.2. The for-in loop on line 5 in the original program in Figure 5.2 iterates through all properties of a Circle object, so the behavior of this loop changes if radius becomes a variable instead of a property. The multiplication in the loop is no longer executed since there is no radius property to be copied. The for-in loop will also copy the drawShape property, but the copied function object will continue to refer to the local variables of the original Circle object that was being copied. As a result, the program would continue to draw circles, but with just half the radius. The ENCAPSULATE PROPERTY refactoring should clearly be disallowed in this case. A JavaScript refactoring tool must carefully take into account how properties are accessed dynamically and prevent ENCAPSULATE PROPERTY in cases where it might lead

---

5 This is not to be confused with the new getter/setter mechanism introduced in ECMAScript 5, which only applies to object literals [22 §11.1.5].
5.2. Motivating Examples

```javascript
function Circle(x, y, r, c) {
    this.x = x;
    this.y = y;
    this.radius = r;
    this.color = c;
    this.drawShape = function(gr) {
        gr.fillCircle(new jsColor(this.color),
                      new jsPoint(this.x, this.y),
                      this.radius);
    }
}

function Rectangle(x, y, w, h, c) {
    this.x = x;
    this.y = y;
    var width = w;
    this.height = h;
    this.color = c;
    this.getWidth = function() {
        return width;
    };
    this.setWidth = function(w) {
        return width = w;
    };
    this.drawShape = function(gr) {
        gr.fillRectangle(new jsColor(this.color),
                        new jsPoint(this.x, this.y),
                        width, this.height);
    }
}
Rectangle.prototype.getArea = function() {
    return this.getWidth() * this.height;
};
```

Figure 5.3: The library of Figure 5.1(a) after applying Encapsulate Property to the `width` property of `Rectangle`.

```javascript
var r1 = new Rectangle(0, 0, 100, 200, 'red');
var r2 = new Rectangle(0, 0, 300, 100, 'blue');
r1.drawShape = r2.drawShape;
drawAll([r1]);
```

Figure 5.4: Alternative client program.

to behavioral changes. In this particular case, a tool could conservatively disallow any of the properties of `Circle` from being encapsulated.

JavaScript allows one to dynamically assign function values to properties, which causes further complications. Suppose that we want to apply Encapsulate Property to the `width` property of `Rectangle` in a situation that includes the library of Figure 5.1(a) and the (artificial) client program of Figure 5.4. The original version of the program draws a red 100-by-200 rectangle. However, if `width` is encapsulated, as shown in Figure 5.3, a red 300-by-200 rectangle is drawn instead. To see why, note that the function stored in property `r1.drawShape` and invoked by `drawAll` comes from `r2.drawShape`, and contains the function originally created during the constructor invocation on line in Figure 5.1. Hence its lexical environment stores the value 300 for `width`, and this is the value read on line in Figure 5.4.
5. Tool-supported Refactoring for JavaScript

(a)

```
var geometry = (function()
{
  function Circle (x, y, r, c) {
    this.x = x;
    this.y = y;
    this.radius = r;
    this.color = c;
    this.drawShape = function (gr) {
      gr.fillCircle(new jsColor(this.color),
      new jsPoint(this.x, this.y),
      this.radius);
    }
  }

  function Rectangle (x, y, w, h, c) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
    this.color = c;
    this.drawShape = function (gr) {
      gr.fillRectangle(new jsColor(this.color),
      new jsPoint(this.x, this.y),
      this.width, this.height);
    }
  }

  Rectangle.prototype.getArea = function () {
    return this.width * this.height;
  }

  return {
    Circle : Circle,
    Rectangle : Rectangle
  };

}());
```

(b)

```
function r(n) { return Math.round(Math.random() * n); }

function drawAll(shapes) {
  var gr = new jsGraphics(document.getElementById("canvas"));
  shapes.map(function(s) { s.drawShape(gr); });
}

var shapes = [];
for (var i = 0; i < 500; i++) {
  var o = new jsColor().rgbToHex(r(255), r(255), r(255));
  switch(r(2)) {
    case 0:
      shapes[i] = new geometry.Circle(r(500),r(500),r(50), o);
      break;
    case 1:
      shapes[i] = new geometry.Rectangle(r(500),r(500),r(50), r(50), o);
      alert(shapes[i].getArea());
      break;
  }
}

drawAll(shapes);
```

Figure 5.5: The example program of Figure 5.1 after applying EXTRACT MODULE to Circle and Rectangle.
5.2. Motivating Examples

The height, on the other hand, is read from property `height` of object `this`; the value of `this` is always the object on which the function is invoked, here `r1`, so `this.height` yields 200.

The problem can be resolved by replacing the identifier reference `width` on line 29 by a call `this.getWidth()`. In Section 5.3, we define the notion of well-scopedness to characterize functions that act as methods of a single object, making it safe to access the encapsulated property directly. Roughly speaking, a function is well-scoped if, on every call, its receiver object is the same as the value that `this` had when the function was defined. In the presence of the client of Figure 5.4, `drawShape` is not well-scoped because of the assignment on line 37. Therefore, our refactoring tool knows that it must replace the identifier reference `width` on line 29 by a call to `this.getWidth`.

5.2.4 Extract Module

JavaScript does not provide language constructs for modularization and relies on a single global name space for all top-level functions and variables, even those that are declared in different files. This is problematic, because it can easily lead to situations where declarations of global variables and functions in one file are clobbered by those declared in another. Fortunately, it is possible to obtain most of the benefits of a module system using closures [18, page 40].

Figure 5.5 shows the example program of Figure 5.1 after applying Extract Module to move the `Circle` and `Rectangle` functions into a new “module” called `geometry`. The basic idea is that these previously global functions become local functions inside an anonymous function, which returns an object literal with properties `Circle` and `Rectangle` through which the functions can be invoked (lines 30–33). This anonymous function is invoked immediately (line 34), and the result is assigned to a newly introduced global variable, `geometry` (line 1). Hence, the constructor functions are now available as `geometry.Circle` and `geometry.Rectangle`. Figure 5.5(b) shows how the client application of Figure 5.1(b) is updated, by using these “qualified names”. Note that this approach has the important benefit that inside the newly introduced closure function, there is no need to refer to the `geometry` variable. For example, the name `Rectangle` on line 26 need not be qualified.

A refactoring tool must take certain precautions when applying Extract Module. For example, observe that choosing the name `shapes` for the new module is problematic because a variable with the same name is already declared on line 42. If we were to perform the refactoring anyway, the shapes “module” would be overwritten, and the constructor calls on lines 48 and 52 would cause runtime errors since the empty array `shapes` does not have properties `Circle` or `Rectangle`.

5.2.5 Discussion

The examples in this section show that refactoring tools for JavaScript have to address a number of challenges that do not arise in statically typed languages such as Java. Chief among these challenges is the lack of static typing and variable declarations, and the use of reflective constructs such as `for-in` loops. We address these challenges with a number of query operations defined on top of a pointer analysis framework. We present
Tool-supported Refactoring for JavaScript

the framework and its queries in Section 5.3 and put them to work in Section 5.4 by specifying the refactorings introduced in this section in more detail.

5.3 A Framework for Refactoring with Pointer Analysis

In this section, we develop the technical machinery needed to precisely specify and implement refactorings like the ones described in the previous section. We first describe a set of basic queries to be provided by an underlying pointer analysis such as the one discussed in Section 5.5. Then, we motivate the analysis questions a refactoring tool needs to answer by taking a closer look at some of the issues illustrated above, and we show how to crystallize them into reusable queries that can be implemented on top of the basic query interface. Section 5.4 will demonstrate how these queries are in turn used to give detailed specifications for several refactorings.

5.3.1 Basic Queries

As the foundation of our framework, we assume a pointer analysis that defines a finite set \( L \) of object labels such that every object at runtime is represented by a label. We assume that \( L \) includes labels to represent environment records [22, §10.2.1]. For technical reasons, we require that if an object label represents an object allocated by a particular \( \text{new} \) expression, then all objects represented by that label are allocated by that expression. Similarly, a single object label cannot represent two function objects associated with different textual definitions.

We say that a set \( L \) of object labels over-approximates a set \( O \) of runtime objects if every object \( o \in O \) is represented by some \( l \in L \). For brevity, we will use the term function definition to mean “function declaration or function expression” and invocation expression to mean “function call expression or new expression”.

The pointer analysis should provide the following queries:

objects For any expression \( e \) in the program, \( \text{objects}(e) \subseteq L \) over-approximates the set of objects to which \( e \) may evaluate, including objects arising from ToObject conversion [22, §9.9]. For a function declaration \( f \), \( \text{objects}(f) \) over-approximates the set of function objects that may result from evaluating \( f \).

scope For any function definition or catch clause \( e \), \( \text{scope}(e) \subseteq L \) over-approximates the set of environment records corresponding to \( e \) at runtime.\(^6\) We additionally define \( \text{scope}(e) := \text{objects}(e) \) for any with expression \( e \).

proto For any object label \( \ell \), \( \text{proto}(\ell) \subseteq L \) over-approximates the possible prototype objects of the runtime objects \( \ell \) represents. We write \( \text{proto}^+(L) \) for the set of transitive prototypes of \( L \subseteq L \) as determined by this query.

props For any object label \( \ell \), \( \text{props}(\ell) \subseteq L \) over-approximates the set of objects that could be stored in properties of \( \ell \) (excluding internal properties).

\(^6\)Observe that \( \text{scope}(f) \) for a function definition \( f \) is not necessarily the same as \( \text{objects}(f) \): the former approximates environment records, the latter approximates function objects.
5.3. A Framework for Refactoring with Pointer Analysis

mayHaveProp, mustHaveProp For any object label $\ell$ and property name $p$, $\text{mayHaveProp}(\ell, p)$ should hold whenever any object represented by $\ell$ may have a property $p$; $\text{mustHaveProp}(\ell, p)$, conversely, should only hold if every object represented by $\ell$ has a property $p$ at all times (for instance if $\ell$ represents an environment record and $p$ is a local variable declared in that environment).

arg, ret For an object label $\ell$ and a natural number $i$, $\text{arg}(\ell, i)$ over-approximates the set of objects that may be passed as the $i$th argument (or the receiver in case $i = 0$) to any function labelled by $\ell$. Similarly, $\text{ret}(\ell)$ over-approximates the set of objects that may be returned from $\ell$.

builtin Given the name $n$ of a built-in object as specified in the language specification [22, §15], $\text{builtin}(n)$ returns the corresponding object label. The object label of the global object will be denoted as $\text{global}$. We also define

\begin{align*}
\text{apply} & := \text{builtin}($\text{Function.prototype.apply}$) \\
\text{bind} & := \text{builtin}($\text{Function.prototype.bind}$) \\
\text{call} & := \text{builtin}($\text{Function.prototype.call}$)
\end{align*}

5.3.2 Visited and Base Objects

Many preconditions deal with name binding. Any refactoring that introduces, renames or removes properties risks causing name capture, i.e., situations where a property expression refers to a different object in the refactored program. Two key concepts are needed when formulating preconditions to avoid name capture: the visited objects of a property expression, and its base objects.

Property lookup in JavaScript is, in most circumstances, prototype based. This means that when evaluating a property expression $e.x$, the property $x$ is first looked up on the object $o_1$ that $e$ evaluates to; if $o_1$ does not have a property of this name, its prototype object $o_2$ is examined, and so on. Eventually, an object $o_n$ is encountered that either has a property $x$, or does not have a prototype object (in which case the lookup returns the $\text{undefined}$ value). We describe this process by saying that the lookup of $e.x$ visits objects $o_1, \ldots, o_n$; if the property is ultimately found on object $o_n$, we call $o_n$ the base object of the lookup.

To see how these concepts are useful for specifying refactorings, consider the case of a refactoring that adds a property $y$ on some object $o$. This refactoring needs to ensure that $o$ is not among the objects that any existing property expression $e.y$ may visit. Otherwise, the base object of an evaluation of that expression could change, possibly altering program behavior.

The usual purpose of adding a new property $y$ to an existing object is to rewrite property expressions that used to resolve to some property $x$ on that object so that they now instead resolve to $y$. For instance, ENCAPSULATE PROPERTY rewrites $\text{this.width}$ on line [26] of Figure 5.1 into $\text{this.getWidth}$ on line [33] of Figure 5.3 to make it resolve to the newly introduced getter function. To prevent the refactored property expression from being resolved with the wrong base object or from overwriting an existing property, we have to require that a lookup of $\text{this.getWidth}$ at this position in the original program would come up empty-handed, that is, that none of the visited objects of the property expression has a property $\text{getWidth}$. This is indeed the case in this example because no property $\text{getWidth}$ is defined anywhere in Figure 5.1.
The same considerations apply to the lookup of local and global variables: global variables are just properties of the global object, while local variables can be viewed as properties of environment records. The concepts of visited objects and base objects can hence be extended to identifier references in a straightforward manner as shown in the accompanying technical report [26].

To underscore this commonality, we introduce the umbrella term *access* to refer to both identifier references (like `r` on line 4 of Figure 5.1) and property expressions, including both *fixed-property* expressions like `s.drawShape` on line 14 of Figure 5.2 and dynamic ones like `nc[a]` on line 4 of Figure 5.2. Identifier references and fixed-property expressions are called *named accesses*.

An over-approximation `possiblyNamed(p)` of all accesses in the program that possibly have name `p` in some execution, and an under-approximation `definitelyNamed(p)` of accesses that definitely have name `p` in every execution can be computed based on purely syntactic information, although pointer analysis may provide additional information that can, e.g., be used to determine that a dynamic property access is always used as an array index and hence cannot have a non-numeric property name.

Given the basic queries introduced in Section 5.3.1, it is not hard to define queries `visited` and `base` to over-approximate visited and base objects of accesses.

For a property expression `e.x`, for instance, `visited(e.x)` can be computed as the smallest set `Lv` satisfying the following two conditions:

1. `objects(e) ⊆ Lv`;
2. if `e.x` is in rvalue position, then for every `ℓ ∈ Lv` with `¬mustHaveProp(ℓ, x)` we must have `proto(ℓ) ⊆ Lv`.

The proviso of the second condition accounts for the fact that deletion of and assignment to properties does not consider prototypes.

The definition of `visited` for identifier references is similar, using `scope` to obtain the relevant environment records.

To over-approximate the set of base objects, we first define a filtered version of `visited` as follows:

\[
\text{visited}(a, x) := \{ \ell ∈ \text{visited}(a) \mid \text{mayHaveProp}(\ell, x) \}
\]

This discards all object labels that cannot possibly have a property `x` from `visited(a)`. For a named access `a` with name `x` in rvalue position, we then define `base(a) := visited(a, x)`, whereas for a dynamic property access or an access in lvalue position we set `base(a) := visited(a)`.

It will also be convenient to have a query `lookup(e, x)` that simulates local variable lookup of an identifier `x` at the position of the expression `e`, and approximates the set of environment records or objects on which `x` may be resolved. This query can be implemented by traversing the function definitions, with blocks and catch clauses enclosing `e`, and then using `scope` and `mayHaveProp` to find possible targets.

---

7The technical report [26] generalizes the concept of accesses even further, but for expository purposes we refrain from doing so here.
5.3. A Framework for Refactoring with Pointer Analysis

5.3.3 Related Accesses

When renaming an access, it is important to determine which other accesses in the program refer to the same property. This is not a well-defined question in general: a given access may at different times be looked up on different base objects and even refer to different property names, so two accesses may sometimes refer to the same property name on the same object, while at other times they do not. In general, we can only determine whether two accesses must *always* refer to the same property, or whether they may *sometimes* do so.

Must-alias information is not very useful for renaming, as explained in Section 5.2: when renaming `this.drawShape` on line 6 of Figure 5.1, we also have to rename `s.drawShape` on line 33, even though it does not necessarily refer to the same property. But if we rename `s.drawShape`, we also have to rename any access that may refer to the same property as that access, viz., `this.drawShape` on line 6 and `this.drawShape` on line 19.

This example suggests that we have to close the set of accesses to rename under the may-alias relation. More precisely, let us call two accesses $a_1$ and $a_2$ *directly related* if their base object may be the same and they may refer to the same property name. The set $\text{related}(a_1)$ of accesses related to $a_1$ is then computed as the smallest set $R$ satisfying the following two closure conditions:

1. $a_1 \in R$;
2. for every $a \in R$, if $a'$ is an access such that $a$ and $a'$ are directly related, then also $a' \in R$.

When renaming $a_1$ we also rename all accesses it is related to. We have argued above why it is necessary to include all related accesses in the renaming. On the other hand, it is also sufficient to just rename these accesses: if any access $a'$ may at runtime refer to the same property as some renamed access $a$, then $a$ and $a'$ are directly related and hence $a'$ will also be renamed. The set of related accesses thus represents a family of properties that have to be refactored together.

5.3.4 Initializing Functions

The Encapsulate Property refactoring looks similar to the Encapsulate Field refactoring for languages like Java and C#, but the very liberal object system of JavaScript allows for subtle corner cases that the refactoring needs to handle. While it is common in JavaScript to make a distinction between normal functions and constructor functions that are only used to initialize newly created objects, this distinction is not enforced by the language.

Any function $f$ can either be invoked through a `new` expression `new f(...)`, in which case the receiver object is a newly created object, or through a function invocation, in which case the receiver object is determined from the shape of the invocation: for an invocation of the form $e.f(...)$, the receiver object is the value of $e$; for an unqualified invocation $f(...)$, the receiver object is usually the global object.

We capture the notion of a function behaving “like a constructor” by saying that a function *initializes* an object $o$ if it is invoked precisely once with that object as its receiver,
5. Tool-supported Refactoring for JavaScript

and this invocation happens before any of o's properties are accessed. For instance, function `Rectangle` in Figure 5.1 initializes all of the objects created on line 44 by invoking `new Rectangle(...)`. 

If a function is only ever invoked using `new` and never invoked reflectively or using a normal function invocation, it obviously initializes all objects created by these `new` invocations. This provides an easy way to approximate the set of objects that are initialized by a function. Let us first define an over-approximation of the set of possible callees of an invocation expression c by `callees(c) := objects(c_f)` where c_f is the part of c containing the invoked expression. Now, given a function definition f, an under-approximation `initializes(f)` of the set of objects that f initializes can be determined by ensuring the following:

1. f is only invoked through `new`, that is
   a) No function/method call c has
      \[ \text{callees}(c) \cap \text{objects}(f) \neq \emptyset. \]
   b) f is not invoked reflectively, i.e.,
      \[ \text{args}(\text{apply}, 0) \cap \text{objects}(f) = \emptyset, \]
      and similarly for `bind` and `call`.
2. For any `new` expression n with
   \[ \text{callees}(n) \cap \text{objects}(f) \neq \emptyset \]
   we have
   \[ \text{callees}(n) \subseteq \text{objects}(f) \]
   This ensures that n definitely calls f.

The first condition ensures that f is invoked at most once on each receiver object, and the second condition ensures that it is invoked at least once. If both conditions hold, f initializes all its receiver objects, so we can set `initializes(f) := \bigcup_{\ell \in \text{objects}(f)} \text{arg}(\ell, 0)`; otherwise, we conservatively set `initializes(f) := \emptyset`.

5.3.5 Well-scopedness

Just as there are no genuine constructors in JavaScript, there are no real methods either. Although it is common to think of a function stored in a property of an object o as a method of o that is only invoked with o as its receiver, this is not enforced by the language, and such a "method" can, in fact, be invoked on any object. As shown in Figure 5.4, this leads to problems when encapsulating properties.

We capture the notion of a function behaving "like a method" by the concept of well-scopedness. A function f is well-scoped in a function g if f is defined within g and whenever an execution of g on some receiver object o evaluates the definition of f, yielding a new function object f_o, then this implies that f_o is always invoked with o as its receiver. If g additionally initializes all objects on which it is invoked, then f is guaranteed to behave like a method on these objects.

To prove that a function definition f is well-scoped in g, as expressed by the query `wellscoped(f, g)`, it suffices to check the following conditions:
5.3. A Framework for Refactoring with Pointer Analysis

```javascript
function A(g) {
  if (g)
    this.f = g;
  else
    this.f = function() {};
}

var a = new A(), b = new A(a.f);
b.f();
```

Figure 5.6: Example program to illustrate the approximation of well-scopedness.

1. \( f \) is a direct inner function of \( g \).

2. \( f \) is only assigned to properties of the receiver of \( g \): whenever the right-hand side \( e_r \) of a simple assignment may evaluate to \( f \) (i.e., \( \text{objects}(e_r) \cap \text{objects}(f) \neq \emptyset \)), the sole intra-procedural reaching definition of \( e_r \) is \( f \) itself, and the left-hand side of the assignment is a property expression of the form \( \text{this}.p \) (for some identifier \( p \)).

3. \( f \) is only invoked on the object in whose property it is stored: any invocation expression \( c \) that may call \( f \) must be of the form \( e.p(\ldots) \), and \( \text{mayHaveProp}(o,p) \) is false for every \( o \in \text{proto}^+(\text{objects}(e)) \).

4. \( f \) is not invoked reflectively (cf. condition 1b in the definition of \( \text{initializes} \)).

The second condition is motivated by considering the example program in Figure 5.6. The function stored in \( a.f \) is not well-scoped in \( A \): the receiver of \( A \) at the point where the function is defined is \( a \), yet when it is called through \( b.f \) the receiver object is \( b \). This non-well-scopedness results from the assignment in line 3 and is detected by condition 2.

5.3.6 Intrinsics and Reflective Property Access

A number of intrinsic properties are treated specially by the runtime system, the browser, or the standard library in JavaScript, for instance the \( \text{length} \) property of array objects or the \( \text{src} \) property of HTML image objects. Refactorings must not attempt to modify these properties. We hence need a query \( \text{intrinsic} \) so that \( \text{intrinsic}(\ell,p) \) holds whenever \( p \) is an intrinsic property on an object labelled by \( \ell \). This query can be defined in terms of \( \text{builtin} \), consulting the relevant standards [22, 92].

Several standard library functions access properties of their argument objects in a reflective way: for instance, \( \text{Object.keys} \) returns an array containing the names of all properties of its argument. To make it possible for refactorings to check for this kind of usage, we need a query \( \text{reflPropAcc} \) such that \( \text{reflPropAcc}(\ell) \) holds whenever a property of an object labelled by \( \ell \) may be accessed reflectively by one of these functions. This query can be defined in terms of \( \text{builtin} \), \( \text{arg} \), \( \text{ret} \) and \( \text{props} \).

Finally, queries \( \text{builtin} \) and \( \text{arg} \) also make it possible to conservatively determine whether a program uses dynamically generated code by checking whether the built-in function \( \text{eval} \) and its various synonyms are ever invoked, and whether the intrinsic property \( \text{innerHTML} \) is assigned to. Our refactoring specifications assume that such a check is performed first and a warning is issued if a use of any of these features has been detected.
5.4 Specifications of Three Refactorings

We will now give detailed specifications of the refactorings Rename, Encapsulate Property and Extract Module that were informally described in Section 5.2.

Each specification describes the input to the refactoring, the preconditions that have to be fulfilled in order for the refactoring to preserve program behavior, and the transformation itself. The preconditions are formulated in terms of the queries introduced in the previous section.

We also provide a brief informal justification of the preconditions.

5.4.1 Rename

Input A named access $a$ and a new name $y$.

Overview The refactoring renames $a$ and its related accesses to $y$.

Definitions Let $B := \bigcup_{r \in \text{related}(a)} \text{base}(r)$; this set labels all objects that are affected by the renaming. Let $x$ be the name part of the access $a$.

Preconditions

1. $x$ is not an intrinsic property on $B$:

$$\forall \ell \in B: \neg \text{intrinsic}(\ell, x)$$

2. Every access to be renamed definitely has name $x$:

$$\text{related}(a) \subseteq \text{definitelyNamed}(x)$$

3. The accesses in $\text{related}(a)$ can be renamed to $y$ without name capture:

$$\forall r \in \text{related}(a): \text{visited}(r, y) = \emptyset$$

In this case, we will also say that $y$ is free for $\text{related}(a)$.

4. $y$ does not cause name capture on $B$, that is:

a) Existing accesses are not captured:

$$\forall r \in \text{possiblyNamed}(y): \text{visited}(r) \cap B = \emptyset$$

b) $y$ is not an intrinsic property on $B$:

$$\forall \ell \in B: \neg \text{intrinsic}(\ell, y)$$

c) Properties of the objects in $B$ must not be accessed reflectively, that is:

i. For any for-in loop with loop expression $e$ it must be the case that $B \cap \text{objects}(e) = \emptyset$.

ii. We must have $\forall \ell \in B: \neg \text{reflPropAcc}(\ell)$. 


5.4. Specifications of Three Refactorings

**Transformation**  Rename every access in $related(a)$ to $y$.

**Justification**  Precondition 2 prevents the renaming if it could affect a computed property access whose name cannot be statically determined.

Preconditions 3 and 4a ensure that accesses in the refactored program resolve to the same property at runtime as in the original program: by 3, an access renamed from $x$ to $y$ is not captured by an existing property $y$; by 4a, an existing access named $y$ is not captured by a property renamed from $x$ to $y$.

Preconditions 1 and 4b ensure that the renaming does not affect properties that have special meaning in the language; for instance, renaming the *prototype* of a function or the *length* property of an array should not be allowed.

Finally, precondition 4c ensures that none of the objects whose properties may be affected by the refactoring have their properties examined reflectively.

### 5.4.2 Encapsulate Property

**Input**  A fixed-property expression $a$.

**Overview**  This refactoring identifies a function $c$ that initializes all base objects of $a$ and its related accesses, and turns the property accessed by $a$ into a local variable of $c$.

Any accesses to the property from within the function $c$ can be turned into accesses to the local variable if they happen from inside well-scoped functions; otherwise they might refer to the wrong variable as seen in Section 5.2. Accesses from outside $c$ are handled by defining getter and setter functions in $c$ and rewriting accesses into calls to these functions.

The preconditions identify a suitable $c$, determine how to rewrite accesses, and check for name binding issues.

**Definitions**  Let $x$ be the name part of $a$, and let $g$ and $s$ be appropriate getter and setter names derived from $x$.

Let $B := \bigcup_{r \in related(a)} base(r)$; this is the set of objects whose properties named $x$ we want to encapsulate.

**Preconditions**

1. There is a function definition $c$ with $B \subseteq initializes(c)$.

   The getter and setter functions are introduced in $c$; since $c$ is invoked on every affected object before any of its properties are accessed, we can be sure that these functions are in place before their first use.

2. The affected objects do not appear on each other’s prototype chains, i.e.,

   $$\neg \exists b_1, b_2 \in B : b_2 \in proto^+(b_1)$$

3. Every access in $related(a)$ is either a fixed-property expression or an identifier reference. (The latter can only happen if a *with* statement is involved.)

4. There is a partitioning $related(a) = A_i \uplus A_g \uplus A_s$ such that:
5. **Tool-supported Refactoring for JavaScript**

a) Every \( a \in A_i \) is of the form `this.x`, it is not an operand of `delete`, and its enclosing function definition \( f \) is well-scoped in \( c \), i.e. `wellscoped(f,c)`.

These are the accesses that will be replaced by identifier references \( x \).

b) No \( a \in A_g \) is in an lvalue position.

These accesses can be turned into invocations of the getter function.

c) Every \( a \in A_s \) forms the left-hand side of a simple assignment.

These accesses can be turned into invocations of the setter function.


6. Naming checks:

a) \( A_i \) can be refactored without name capture:

\[
\forall a \in A_i : \text{lookup}(a,x) \subseteq \{ \text{global} \}
\]

b) The declaration of the new local variable \( x \) in \( c \) does not capture existing identifier references.

\[
\forall a \in \text{possiblyNamed}(x) : \text{visited}(a) \cap \text{scope}(c) = \emptyset
\]

c) \( x \) is not an intrinsic property on \( B \):

\[
\forall \ell \in B : \neg \text{intrinsic}(\ell,x)
\]

7. If \( A_g \neq \emptyset \) then \( g \) must be free for \( A_g \) and must not cause name capture on `initializes(c)` (cf. preconditions [3] and [4] of `Rename`). Similarly, if \( A_s \neq \emptyset \) then \( s \) must be free for \( A_s \) and must not cause name capture on `initializes(c)`.

**Transformation** Insert a declaration `var x` into \( c \). Insert a definition of the getter function into \( c \) if \( A_g \neq \emptyset \), and similarly for \( A_s \) and the setter function. Replace accesses in \( A_i \) with \( x \), accesses in \( A_g \) with invocations of the getter, in \( A_s \) with invocations of the setter.

**Justification** This refactoring converts properties of objects into bindings in environment records. The preconditions ensure that property accesses can be rewritten into accesses to the corresponding local variable binding, while preventing any changes to other accesses to properties or local variables that do not participate in the refactoring.

Consider a runtime object \( o \) labeled by some \( \ell \in B \). By condition [4], there is precisely one invocation of \( c \) on \( o \), which creates an environment record \( \rho_o \). In the refactored program, this environment record contains a binding for a local variable \( x \), which is captured by the getter and setter functions stored in properties \( g \) and \( s \) of \( o \).

Consider now a property access \( a_x \) in the original program that accesses property \( x \) of object \( o \). This means that \( a_x \in \text{related}(a) \), so condition [4] ensures that \( a_x \) is in one of \( A_i \), \( A_g \) and \( A_s \). In the two latter cases, the property access will be rewritten into an invocation of the getter method \( g \) or the setter method \( s \) on \( o \).

By condition [7], this invocation will not be captured by another method of the same name, and by condition [2] it will not be captured by the accessor methods of another
refactored object. By condition [4] $g$ and $s$ are already defined, and by condition [7] they are guaranteed not to have been overwritten in the meantime, hence the accessor functions set up by $c$ are executed, accessing the correct binding in $\rho_o$.

If $a_x \in A_i$, the property access is refactored to a simple identifier reference $x$. We know from condition [4a] that $a_x$ must occur in some function definition $f$, which is well-scoped in $c$, and that it must be of the form $\text{this.}x$. Hence $f$ is, in fact, invoked with $o$ as receiver, which by the definition of well-scopedness means that the invocation of $c$ whose bindings are captured by $f$ also has receiver $o$. In other words, $f$ captures the bindings of $\rho_o$. Condition [6a] ensures that the identifier reference $x$ in the refactored program is not captured by any other local variable, and hence accesses the binding of $x$ in $\rho_o$ as desired.

The requirement about $a_x$ not being an operand of delete is purely technical: local variable bindings cannot be deleted in JavaScript.

Since the set of properties of $o$ has changed in the refactored program, any code that reflectively accesses properties of $o$ or the set of property names of $o$ may change its behavior; conditions [3], [5] and [6c] guard against this. Finally, condition [6b] ensures that no existing local variable bindings are upset by the newly introduced local variable $x$ in $c$.

**Remarks** Note that condition [4] makes it impossible to refactor accesses like $++e.x$ that both read and write the encapsulated property, unless they can be replaced by an identifier reference. It is straightforward to extend the refactoring to take care of such accesses, at the cost of a slightly more complicated transformation involving both getter and setter invocations in the same expression [26].

### 5.4.3 Extract Module

**Input** Contiguous top-level statements $s_1, \ldots, s_m$ containing a set $P = \{p_1, \ldots, p_n\}$ of identifiers to extract and an identifier $M$ to be used as module name.

**Overview** The global variables $p_1, \ldots, p_n$ are turned into properties of a newly declared global module variable $M$. Schematically, the transformation performed by the refactoring is as follows:

```javascript
var M = (function() {
    var $p_1$, ..., $p_n$;
    $s_1$; ... $s_m$;
    return {
        $p_1$: $p_1$, ..., $p_n$: $p_n$
    };
})();
```

We refer to the code defining $M$ as the module initialization code. To reason about the correctness of the transformation, it is helpful to partition program execution into three phases: before, during and after execution of the initialization code. Being a top-level statement, the module initialization code is executed only once.

None of the variables in $P$ must be accessed before module initialization since the module $M$ containing them has not been defined yet. After module initialization, on the other hand, they can be accessed as properties of $M$, i.e., $M.p_1, \ldots, M.p_n$. It is clearly not possible to access them in this way during module initialization ($M$ is, after all, not defined yet), but we can instead access the corresponding local variables $p_1, \ldots, p_n$ if they are in scope.
Closures created during module initialization may still be able to access a local variable even after module initialization has finished. This should, however, be avoided unless it can be proved that the variable is never assigned to after module initialization: if not, the local variable \( p \) and the property \( M.p \) may have different values, which could change program behavior.

The preconditions determine a set \( Q \) of accesses that have to be converted into property accesses of the form \( M.p \), and a set \( U \) of accesses that can use the local variables of the module. The preconditions also prevent access to module variables before initialization and name binding issues.

Definitions Let \( S \) be the set of all accesses appearing in the statements \( s_1, \ldots, s_m \), and let \( I \subseteq S \) be the accesses that are not nested inside functions. Accesses in \( I \) are thus guaranteed to only be evaluated during module initialization.

Let \( I^* \) be an over-approximation of the set of all accesses that may be evaluated before or during module initialization. This can be obtained by building a transitive call-graph of all top-level statements up to \( s_m \), using query callees to determine possible callees of invocations. Finally, let \( C \) contain all accesses in the program except those in \( I^* \). Accesses in \( C \) are thus guaranteed only to be evaluated after module initialization is complete.

For \( p \in P \), we define \( A_p \) to be the set of accesses that may refer to the global variable \( p \), and \( A_P := \bigcup_{p \in P} A_p \). We define \( \text{mutable}(p) \) to hold if \( A_p \) contains a write access that does not belong to \( I \), i.e., if \( p \) may be written after module initialization is complete.

Preconditions

1. Any access that may refer to some property in \( P \) must refer to that property, i.e., for every \( p \in P \) and \( a \in A_p \):

\[
    a \in \text{definitelyNamed}(p) \land \text{visited}(a, p) = \{\text{global}\}
\]

2. There is a partitioning \( A_P = Q \sqcup U \) as follows:

   a) \( Q \subseteq C \)
   
   b) \( M \) is free for every \( q \in Q \) (cf. precondition \( \Box \) of Rename).
   
   c) For every \( u \in U \) referring to \( p \in P \), the following holds:

      i. \( u \in I \lor (u \in S \land \neg \text{mutable}(p)) \)
      
      ii. \( u \) is an identifier reference.
      
      iii. \( \text{lookup}(u, p) \subseteq \{\text{global}\} \).

3. \( M \) does not cause name capture on \( \text{global} \) (cf. precondition \( \blacksquare \) of Rename).

4. No \( p \in P \) is an intrinsic on \( \text{global} \):

\[
    \forall \ell \in B: \neg \text{intrinsic} (\ell, p)
\]

Transformation Replace \( s_1, \ldots, s_m \) with the definition of module \( M \) as shown above; qualify accesses in \( Q \) with \( M \).
5.4. Specifications of Three Refactorings

Figure 5.7: The refactoring plug-in for Eclipse. The user has attempted to rename `Vector.prototype.removeFirst` to `remove`, which the tool correctly determines would clash with an existing property of the same name.

**Justification** The refactoring introduces a new global variable $M$ and removes the global variables $p_1, \ldots, p_n$. Condition 3 ensures that no existing access to a variable $M$ is captured by the newly introduced module variable, and that the set of global variables is not examined reflectively. Condition 4 ensures that none of the global variables to be modularized has special semantics. It should, for instance, be impossible to extract the global variable `window` into a module.

The remaining preconditions ensure that accesses to global variables $p_1, \ldots, p_m$ can be consistently refactored. Condition 1 requires that any access either must definitely refer to some $p \in P$, or must not refer to any variable in $P$. Condition 2a checks that accesses in $Q$, which are to be qualified with a reference to $M$, are only evaluated after the module is defined. For the same set of accesses, condition 2b ensures that the reference to $M$ that will be inserted by the refactoring cannot be captured by an existing variable $M$.

Finally, condition 2c makes sure that every access $u \in U$, which used to refer to one of the global variables $p \in P$, can directly access the local variable this variable has been turned into. Sub-condition 2(c)i requires that $u$ is either only evaluated during module initialization, or that it refers to an immutable module member and is lexically nested within the module definition. Either way it can access module members without qualification. Sub-condition 2(c)ii rules out the somewhat subtle case of an access of the form $e.p$, where $e$ evaluates to the global object, but may have side effects; such an access cannot simply be turned into an identifier reference $p$, as this would suppress the side effects of $e$. Sub-condition 2(c)iii ensures that no existing local variable will capture the refactored access $u$.
5. Tool-supported Refactoring for JavaScript

5.5 Implementation

We have implemented a refactoring tool in Java that offers the refactorings described in Section 5.4. The tool is integrated as a plug-in into Eclipse as shown in Figure 5.7. In this section, we will describe the pointer analysis that underlies the implementation of the framework that we presented in Section 5.3.

We first derive a flow graph from the source code of the original program, similar to the one used in the TAJS program analysis [46]. From this flow graph, we create a def-use graph that abstracts away control flow and with statements. We then run a pointer analysis using standard techniques, with lattice and constraints that are reminiscent of the ones used in Gatekeeper [36] (although without using Datalog). The use of a def-use graph captures a small amount of flow sensitivity, similar to what SSA-form has been shown to contribute to a flow-insensitive analysis [42].

For context sensitivity, we experimented with both k-CFA and object sensitivity (i.e., using the value of this as the context), and found object sensitivity to be the most effective. The analysis uses heap specialization (i.e., some object labels include a context component) and a simple widening function to ensure termination when combined with object sensitivity.

To obtain a useful modeling of arrays, we introduce a special property name NumberProperty representing all properties whose name is a number (i.e., array entries). For dynamic property expressions where the property name is definitely a number, the analysis reads/writes the NumberProperty of the receiver; otherwise, it conservatively reads/writes all of its properties.

Several built-in functions (such as call and apply) are supported by means of specialized transfer functions. All other built-in functions are modelled by simple JavaScript mock-up functions that we include in the analysis.

We model the HTML DOM and some other browser features using a special object label DOM. Some global variables, such as document, are initialized to refer to DOM. Moreover, we conservatively assume that (1) any property of DOM may point to DOM, (2) any function reachable from DOM may be invoked with DOM as the this argument and any number of actual arguments that all may point to DOM, and (3) if DOM is invoked as a function, it stores all its arguments as properties on DOM, and returns DOM. Rules 2 and 3 together take care of event handlers being registered on HTML elements. We avoid many of the challenges that arise with the more detailed modeling used in TAJS [48] by using a relatively simple abstract domain.

Given this basis, the queries of the framework of Section 5.3 are straightforward to implement, as are the refactorings themselves.

5.6 Evaluation

To gain some insight into the practical applicability and usefulness of our approach, we have evaluated our refactoring tool on a collection of existing JavaScript programs.

In situations where the tool determines that a requested refactoring can be performed, the refactoring preconditions ensure that it is safe to perform the refactoring, without

---

8Note that this is purely a UI-level integration; the underlying analysis and the code for precondition checking and program transformation is independent of Eclipse.
5.6. Evaluation

When a refactoring attempt is rejected by the tool, either the refactoring would in fact change the behavior of the program, in which case the answer given by the tool is correct, or the rejection is caused by the analysis being too conservative. In the latter case, the imprecision may be in the refactoring preconditions that are defined in terms of our queries (Section 5.4), in the definition of the derived queries on top of the basic ones (Section 5.3.2–5.3.6), or in the underlying pointer analysis that we employ to implement the basic queries (Section 5.5). To quantify how often these situations occur, we aim to answer these research questions:

**Q1:** How often is a refactoring rejected because its preconditions are too conservative?

**Q2:** How often is a refactoring rejected because a derived query is defined too conservatively?

**Q3:** How often is a refactoring rejected because of imprecision in the underlying pointer analysis?

**Q4:** How often does our RENAME refactoring give a different outcome than syntactic search-and-replace as performed in syntax-directed editors?

We collected a suite of benchmark programs and designed a set of experiments for each of the refactorings specified in Section 5.4 to evaluate them with regard to these questions.

Table 5.1 shows an overview of our evaluation results, explained in more detail below: for every refactoring, the table shows the total number of attempted refactorings on our benchmarks in column “total trials”, with the number of successful trials in the next column; we partition the set of rejected trials according to our research questions into cases where overly strict preconditions prevented the application of an otherwise unproblematic refactoring, cases where imprecise derived queries were an obstacle, cases where the underlying pointer analysis itself was at fault, and finally cases where the rejection was indicative of a real danger of unsoundness.

We will now first give an overview of our benchmark collection, then present detailed evaluation results for each of the refactorings, and finally summarize our findings by answering the research questions.

<table>
<thead>
<tr>
<th>refactoring</th>
<th>total trials</th>
<th>successful trials</th>
<th>rejected trials</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>total</td>
<td>imprecise</td>
<td>imprecise</td>
<td>imprecise</td>
</tr>
<tr>
<td>Rename</td>
<td>10612</td>
<td>10693</td>
<td>5919</td>
<td>0</td>
<td>0</td>
<td>669</td>
</tr>
<tr>
<td>Encapsulate Property</td>
<td>510</td>
<td>363</td>
<td>147</td>
<td>35</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>Extract Module (1)</td>
<td>50</td>
<td>43</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extract Module (2)</td>
<td>15</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Quantitative evaluation of our refactoring tool.
5. Tool-supported Refactoring for JavaScript

5.6.1 Benchmark Programs

We have gathered 50 JavaScript programs. Four are taken from the V8 benchmarks\(^9\), 23 from Chrome Experiments\(^{10}\), 18 from the 10K Apart Challenge\(^{11}\), and 5 from IE Test Drive\(^{12}\). When collecting these benchmarks, we explicitly excluded programs that our pointer analysis cannot analyze in a few minutes and ones that use non-trivial dynamic code execution (e.g., using `eval`). Four of the benchmarks use trivial dynamic code, such as `setTimeout("loop();", 50)` which we have manually replaced by the more manageable variant `setTimeout(loop, 50)`. For 27 of the benchmarks, the tool produces a warning that they may contain assignments to the `innerHTML` property of a DOM object, which can potentially be used to run dynamically generated code, however manual inspection revealed that this is not the case in any of the programs.

Each benchmark comprises between 300 and 1700 lines of JavaScript code, and all perform non-trivial tasks. On a 3.0 GHz PC, each benchmark is analyzed in less than 4 seconds using 256 MB memory. The time required for refactoring-specific computations is negligible compared to the time taken by the pointer analysis.

5.6.2 Rename

Our `Rename` refactoring can rename both local variables and properties. Local variables are trivial to rename since there are no `with` statements in our benchmarks, so we focus on renaming of properties.

We have systematically applied our refactoring to every property expression and property initializer in each benchmark, with the aggregate results shown in Table 5.1 in the row labeled `Rename`. Out of a total of 16612 attempted rename operations, 10693 were successfully applied, and 5919 were rejected by our tool. Further analysis revealed that of these rejections, 5250 were justified. Two benchmarks are responsible for the remaining 669 rejections. In `raytracer` from Chrome Experiments, there are 1062 renamable accesses but 665 of these are wrongly rejected, due to the pointer analysis being imprecise. In `flyingimages` from the IE Test Drive benchmarks, the program adds some custom properties to a DOM element, which due to our imprecise DOM model are then assumed to be intrinsic; thus our tool refuses to rename these properties. The remaining 48 benchmarks do not give rise to any `Rename`-specific spurious warnings.

To evaluate how our tool compares to a simple search-and-replace performed at the level of the abstract syntax tree (AST) in a syntax directed editor, we use the equivalence classes defined by the `related` query to divide all the accesses in a benchmark into components. Accesses in a single component always get renamed together. Our tool distinguishes itself from simple search-and-replace tools when different components contain accesses with the same name. In particular, our tool will rename a smaller set of accesses than search-and-replace would, and if one component can be renamed while another cannot (e.g., an access in it may refer to an intrinsic property), search-and-replace would change the program’s behavior, whereas our approach would reject the refactoring.

The tool finds that 28 of the 50 benchmarks contain multiple renamable components with the same name, and 19 contain same-name components where some can be renamed.
5.6. Evaluation

and others are correctly rejected (18 benchmarks fall into both categories). Overall, our tool succeeds in renaming 1567 components, with 393 of them having a name in common with another component in the same program. This indicates that our RENAME refactoring will often be more precise than search-and-replace in practice.

To summarize, RENAME leads to smaller source code transformations than search-and-replace in about 25% of the cases. Of the refactoring attempts that were not justifiably rejected, it issues spurious warnings in only 6% of the cases. The spurious warnings are all caused by imprecision in the pointer analysis.

5.6.3 Encapsulate Property

We have exhaustively applied the Encapsulate Property refactoring to every property expression of the form `this.x` appearing in an lvalue position inside a function that is invoked at least once in a new expression, with the results shown in Table 5.1 in the row labeled Encapsulate Property.

In the 50 benchmarks, there are 510 such expressions. Our tool is able to successfully encapsulate 363 of them, ignoring warnings about assignments to `innerHTML`. In the remaining 147 cases, the tool reports a precondition failure and rejects the refactoring.

For 82 of these cases, the rejection is justified: in three cases, getter/setter methods already exist; in eight cases the encapsulated property would shadow references to a global variable; in the remaining 71 cases there is a name clash with a parameter or local variable of the enclosing function. We manually verified that these cases can be refactored successfully if the naming conflict is first resolved by renaming.

Of the 65 remaining cases, where the refactoring is rejected although it should have been successful, 35 are due to a limitation of our specification of Encapsulate Property: it requires all objects on which the property is encapsulated to be initialized by the same function. In some cases, however, there are identically named properties on objects constructed by different constructors, which need to be encapsulated at the same time because there are accesses that may refer to either property. Supporting this situation seems like a worthwhile extension.

Finally, there are 30 cases where the pointer analysis yields imprecise results that cause spurious precondition violations. Of these, 19 cases could be fixed by improving the modelling of standard library array functions.

The concept of well-scopedness and the conservative analysis to determine well-scopedness described above prove to be adequate on the considered benchmarks: there are 28 cases where properties to be encapsulated are accessed from within an inner function of the constructor, and in all cases the analysis can establish well-scopedness, allowing the access to be replaced by an identifier reference instead of a getter invocation.

In summary, our tool is able to handle about 85% of the encapsulation attempts satisfactorily (not counting the justifiably rejected attempts). The remaining 15% are caused by, in about equal parts, restrictions of the specification and imprecision of the pointer analysis.

5.6.4 Extract Module

The Extract Module refactoring is difficult to evaluate in an automated fashion, since appropriate module boundaries have to be provided for every benchmark. We have per-
formed two sets of experiments. In the first experiment, we extracted, for every benchmark, the code in each HTML `script` element into its own module; in the case of stand-alone benchmarks we chose source files as the unit of modularization instead. The results of this experiment are shown in Table 5.1 in the row labeled `Extract Module` (1). In the second experiment, we manually determined a suitable modularization for a subset of our benchmarks and used our tool to perform it; again, the results are shown in Table 5.1 in row `Extract Module` (2).

For the first experiment, the automated modularization was successfully performed on 43 out of 50 benchmarks. On the remaining seven benchmarks, the refactoring was rejected since they contain accesses to module members for which the refactoring cannot prove that they either definitely happen only during module initialization, or definitely happen only after initialization. These rejections turn out to be justified: the accesses in question are performed by event handlers registered before or during module initialization. While it is highly likely that these handlers will not fire until after initialization is complete, this is not guaranteed.

In three cases, the rejections are arguably due to the very coarse module structure imposed by this experiment. If the code that installs the event handlers is excluded from the module, the handlers are guaranteed to only fire after initialization and the refactoring can go ahead. In the remaining four benchmarks, on the other hand, event handlers are installed through HTML attributes before the handler functions are even defined, which could potentially cause races even in the original program.

For the second experiment, we randomly selected 15 benchmarks that are not already modularized and whose global variables have sufficiently descriptive names to make it easy to manually determine a possible modularization. In three of these programs, we took comments into account that already suggested a functional grouping of global functions. Our tool can perform the proposed modularization on 11 of the 15 benchmarks. The remaining four are again rejected due to potential races on event handlers.

In both experiments, our tool was thus able to handle all test cases correctly. The categorization of accesses according to whether they are evaluated before or after module initialization proved to be a valuable aid in detecting potentially subtle bugs that could be introduced by the refactoring.

5.6.5 Summary

Overall, the results of our evaluation are promising. Most attempted refactorings are performed successfully, and when our tool rejects a refactoring it mostly does so for a good reason. We briefly summarize our findings and answer the general research questions posed at the beginning of this section.

Q1: Rejections due to rigid preconditions  Spurious rejections resulting from overly conservative preconditions are not very common: this happens in 35 out of 510−82 applications (8.2%) of `Encapsulate Property`, and not at all for `Rename` and `Extract Module`.

Q2: Rejections due to derived queries  The derived queries are always sufficiently precise in our experiments. For instance, `Encapsulate Property` needs to prove well-
scopedness for 28 functions, and all of them are indeed shown to be well-scoped by the algorithm described in Section 5.3.5.

**Q3: Rejections due to imprecise pointer analysis** Spurious rejections resulting from imprecision of the pointer analysis occur occasionally: 669 of 16612−5250 applications (5.9%) of Rename and 30 of 510−82 applications (7.0%) of Encapsulate Property are rejected for this reason; and none for Extract Module.

**Q4: Improvement over naive search-and-replace** For 393 out of 1567 groups of accesses that must be renamed together (25%), Rename avoids some of the unnecessary modifications performed by AST-level search-and-replace.

These results indicate that the precision of the refactoring preconditions, the derived queries, and the pointer analysis is sufficient for practical use, and that our technique has advantages in practice compared to naive approaches.

### 5.6.6 Discussion

The validity of our evaluation may be threatened by (1) benchmark selection, (2) analysis limitations, and (3) selection of refactoring targets.

While we only consider a relatively small number of benchmarks of modest size, the programs included do demonstrate a variety of application areas, from the more numerically oriented V8 benchmarks to browser-based games and visualization programs in the other benchmark sets. They also exhibit very different programming styles, with some benchmarks making heavy use of the object system and others written in an entirely procedural style.

One notable feature of all our benchmarks is that none of them make use of a framework library such as jQuery, Prototype, or MooTools. The pointer analysis currently used in our implementation cannot tackle such libraries due to scalability issues. It is possible that the meta-programming techniques employed by some of these frameworks could lead to very imprecise analysis results that may lead to a large number of spurious rejections. In this case, it could be worthwhile to extend the analysis with special knowledge about particularly tricky framework functions.

Our analysis has certain limitations that may affect the validity of our results. In particular, our implementation only analyzes code that is reachable either from top-level statements or from the DOM. Other code does not influence the refactoring and is itself not affected by refactoring. This means that our tool cannot safely be applied to library code alone, since most of the functions in a library will be considered dead code when there is no client to invoke them. For statically typed languages, this problem can be sidestepped by assuming, for instance, every method to be an entry point, with the parameter types providing a conservative approximation of the possible points-to sets of arguments. This is not easy to do in JavaScript, and making worst-case assumptions about argument values would lead to unacceptable precision loss. All of our benchmarks are standalone applications, yet about half of them contained some amount of unused code. This indicates that the issue may indeed deserve further attention.

As a second restriction, our analysis currently does not attempt to analyze dynamically generated code. We handle this in our refactoring tool by issuing a warning if a potential
use of such code is detected to alert the user of possible changes to the behavior of the program.

Finally, our pointer analysis does not currently model ECMAScript 5 getter and setter properties on object literals, but these are not used in the benchmarks anyway.

These shortcomings of the analysis, however, do not seriously jeopardize the validity of our approach, since we have been careful to introduce a clean separation between analysis and refactoring by means of the framework described in Section 5.3. This makes it easy to plug in a more powerful pointer analysis without having to change the specifications or implementations of the refactorings themselves.

As a final threat to validity, one might question the selection of targets on which to apply our refactoring tool. We have based our evaluation on exhaustively applying the refactorings to as many targets in the code as possible to avoid selection bias. Many of these applications would most likely not make sense in an actual development context; it is hence not clear what percentage of spurious rejections a user of our tool would experience in practice. However, the overall percentage of spurious rejections in our experimental evaluation is so low as to make it seem likely that our tool would behave reasonably in practice.

5.7 Related Work

Two broad categories of related work can be distinguished: previous work on refactoring in general, and work on static analysis of JavaScript programs.

5.7.1 Refactoring

The field of refactoring started in the early 1990s with the Ph.D. theses of Opdyke [63] and Griswold [34]. Since then, the refactoring community has focused on developing automated refactoring tools for both dynamically typed languages (e.g., [61, 70]), and for statically typed languages (e.g., [31, 83, 89]). The discussion below will focus on previous work on refactoring for dynamically typed languages.

Work by Roberts et al. on the Refactoring Browser [70] targets Smalltalk, a dynamically typed language in which some of the same challenges addressed in this paper arise. For method renaming, e.g., it becomes difficult or impossible to determine statically which call sites need to be updated in the presence of polymorphism and dynamically created messages. Unlike our approach, which is based on static pointer analysis, Roberts et al. adopt a dynamic approach to this problem, in which renaming a method involves putting a method wrapper on the original method. As the program runs, the wrapper detects sites that call the original method and rewrites those call sites to refer to the renamed method instead. The main drawback of this approach is that it relies on a test suite that exercises all call sites to be rewritten.

The Guru tool by Moore [61] provides automatic refactoring for the Self programming language. Guru takes a collection of objects, which need not be related by inheritance, and restructures them into a new inheritance hierarchy in which there are no duplicated methods, in a way that preserves program behavior. Moore’s algorithm is based on a static analysis of the relationship between objects and methods in the system. Unlike our work, Moore’s approach does not involve the programmer in deciding what refactorings to apply and where to apply them.
Refactoring support in IDEs for JavaScript appears to be in its infancy. Eclipse JSDT [21] and the JetBrains JavaScript Editor [30] aim to provide refactoring support for JavaScript, but the current implementations are fairly naive. RENAME in the JavaScript Editor, for instance, seems to essentially just perform search-and-replace on the AST. Renaming property x in Circle in the example of Figure 5.1, for instance, would also rename all properties with name x in the jsDraw2D library that the program uses.

Two projects at the IFS Institute for Software focused on developing JavaScript refactoring plug-ins for Eclipse JSDT [8,11], but their results do not seem to have been published and are not currently available.

5.7.2 Analysis for JavaScript

Several authors have pursued forms of static program analysis for JavaScript. The TAJS analysis tool by Jensen et al. [46–48] aims at detecting common programming errors in JavaScript programs. Anderson et al. [6] define a type system for a core calculus based on JavaScript along with an associated constraint-based type inference algorithm. Jang and Choe [45] use a constraint-based analysis for optimizing programs written in a restricted variant of JavaScript. The Gatekeeper tool by Guarnieri and Livshits [36] and the Actarus tool by Guarnieri et al. [38] use static analysis to enforce security policies in JavaScript programs, e.g., that a program may not redirect the browser to a new location or that untrusted information cannot flow to sensitive operations. Guha et al. [40] describe a core calculus for JavaScript and use that formalism to design a type system that statically ensures a form of sandboxing. Other work by Guha et al. [39] involves a k-CFA analysis for extracting models of client behavior in AJAX applications. The Kudzu tool by Saxena et al. [73] performs symbolic execution on JavaScript code and uses the results to identify vulnerability to code injection attacks.

Like our work, many of these analyses rely heavily on the results of a pointer analysis. For example, the TAJS tool performs a pointer analysis as part of its analysis, the optimization technique by Jang and Choe relies directly on pointer analysis, and Gatekeeper’s security policies are expressed in terms of a Datalog-based pointer analysis. In all of these instances, the pointer analysis provides may-point-to information, similar to the underlying analysis in our refactoring framework. However, as we have illustrated in Sections 5.2 and 5.3 may-point-to information does not directly provide a useful abstraction for sound refactorings in JavaScript, which has motivated the higher-level concepts that appear as queries in our framework, such as the notions of relatedness and well-scopedness.

5.8 Conclusion

We have presented a principled approach for tool-supported refactoring for JavaScript programs. The key insight of our work is that—despite the challenging dynamic features of the JavaScript language—it is possible to capture fundamental correctness properties of JavaScript refactorings using a small collection of queries in a framework based on pointer analysis. With this framework, we have demonstrated that the complex preconditions of refactorings, such as RENAME, ENCAPSULATE PROPERTY and EXTRACT MODULE, can be expressed in a concise manner. Our experiments show that the refactoring preconditions we formulate have high accuracy. Most importantly, if a programmer’s request to perform
a refactoring is rejected by the tool, it is usually because the refactoring would in fact change the behavior of the program.

In future work, we will focus on advancing the scalability of the underlying pointer analysis, and we plan to provide specifications and implementations of other refactorings by way of our framework. Another direction of work is to adapt our techniques to other dynamically typed languages.
Chapter 6

Efficient Construction of Approximate Call Graphs for JavaScript IDE Services

This chapter is a copy of an ICSE’13 article [23] by Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip.

Abstract

The rapid rise of JavaScript as one of the most popular programming languages of the present day has led to a demand for sophisticated IDE support similar to what is available for Java or C#. However, advanced tooling is hampered by the dynamic nature of the language, which makes any form of static analysis very difficult. We single out efficient call graph construction as a key problem to be solved in order to improve development tools for JavaScript. To address this problem, we present a scalable field-based flow analysis for constructing call graphs. Our evaluation on large real-world programs shows that the analysis, while in principle unsound, produces highly accurate call graphs in practice. Previous analyses do not scale to these programs, but our analysis handles them in a matter of seconds, thus proving its suitability for use in an interactive setting.

6.1 Introduction

Over the past decade, JavaScript has turned from a niche language for animating HTML pages into an immensely popular language for application development in many different domains. Besides being the enabling technology for Web 2.0 applications such as Google Mail, it is becoming a popular choice for server-side programming with Node.js, for writing cross-platform mobile apps with frameworks like PhoneGap, and even for implementing desktop applications.

This increase in popularity has led to a demand for modern integrated development environments (IDEs) for JavaScript providing smart code editors, software maintenance tools such as automated refactorings, and code analysis tools. While a variety of mature IDEs exist for languages like Java and C#, such tools have only just begun to appear
6. Approximate Call Graphs for JavaScript IDE Services

for JavaScript: existing IDEs such as Eclipse, Komodo IDE and NetBeans are starting to support JavaScript, while new IDEs specifically targeted at web programming such as WebStorm and Cloud9 are also gaining traction.

Compared with their Java counterparts, however, these IDEs are fairly bare-bones. Code navigation and completion use heuristics that sometimes fail unexpectedly, while refactoring and analysis is all but unsupported. There has been some work on more principled tools and analyses [27, 36, 38, 46], but these approaches do not yet scale to real-world applications.

A key impediment to advanced tooling for JavaScript is the difficulty of building call graphs, that is, determining the functions that a given call may invoke at runtime. Such reasoning is required in IDEs both for basic features like “Jump to Declaration”, and also for refactoring or analysis tools that need to reason interprocedurally.

To be useful in an IDE, a call graph construction algorithm must be lightweight and scalable: modern programmers expect IDE services to be constantly available, so it should be possible to quickly compute call graph information on demand, even for large, framework-based JavaScript applications.

Demands on precision and soundness may vary between clients: for Jump to Declaration, listing many spurious call targets is perhaps even worse than occasionally missing one, while analysis tools may prefer a fairly conservative call graph. Achieving absolute soundness is, however, almost impossible in JavaScript due to widespread use of eval and dynamic code loading through the DOM. A useful compromise could be an unsound call graph construction algorithm that can, to some extent, quantify its degree of unsoundness, for instance by indicating call sites where some callees may be missing; the IDE can then pass this information on to the programmer.

In Java, call graphs can be efficiently constructed using class hierarchy analysis [19], which uses type information to build a call graph. However, since JavaScript is dynamically typed and uses prototype-based inheritance, neither class hierarchy analysis nor its more refined variants that keep track of class instantiations [9, 84] are directly applicable. Additionally, these analyses cannot easily handle first-class functions.

An alternative are flow analyses such as CFA [77] or Andersen’s points-to analysis [5] that statically approximate the flow of data (including first-class functions) to reason about function calls. While typically not fast enough for interactive use, such analyses could still be used for non-interactive clients like analysis tools. However, while state-of-the-art flow analyses for JavaScript can handle small framework-based applications [80], they do not yet scale to larger programs.

In this work, we present a lightweight flow analysis specifically designed to efficiently compute approximate call graphs for real-world JavaScript programs. Its main properties are:

1. The analysis is field-based [43, 57], meaning that it uses a single abstract location per property name. Thus, two functions that are assigned to properties of the same name will become indistinguishable as call targets.

2. It only tracks function objects and does not reason about any non-functional values.

3. It ignores dynamic property accesses, i.e., property reads and writes using JavaScript’s bracket syntax.
These design decisions significantly reduce the number of abstract locations and objects, thus dramatically improving scalability. While precision could, in principle, suffer, we show in our evaluation that this does not happen in practice.

Clearly, ignoring dynamic property reads and writes makes the analysis unsound, but this is a consequence of the first two design decisions: since we only track function objects, we cannot reason about the possible string values of $p$ in a dynamic property access $e[p]$, and using a field-based approach means that imprecision cannot be contained by reasoning about aliasing. However, previous work has shown that many dynamic property accesses are correlated [80], i.e., they copy the value from property $p$ in one object to property $p$ of another. With a field-based approach, such a copy is a no-op, since the analysis uses a single representation for all $p$ properties anyway. Our evaluation indicates that in practice, very few call targets are missed due unsoundness.

Like any flow analysis, our analysis faces a chicken-and-egg problem: to propagate (abstract) argument and return values between caller and callee we need a call graph, yet a call graph is what we are trying to compute. We explore two analysis variants that tackle this problem in different ways.

The first is a standard optimistic analysis in the terminology of Grove and Chambers [35] that starts out with an empty call graph, which is gradually extended as new flows are discovered until a fixpoint is reached.

The second variant is a pessimistic analysis that does not reason about interprocedural flow at all and simply gives up on call sites whose call target may depend on such flow, except in cases where the callee can be determined purely locally.

We have implemented both of our techniques and performed an extensive empirical evaluation on ten large, real-world JavaScript web applications, many of them based on popular frameworks. To show the feasibility of using our analyses in an IDE, the implementation operates on abstract syntax trees (ASTs) as IDE-based tools normally do, rather than on an intermediate code representation as is typical for flow analyses.

Both analyses scale very well and are able to build call graphs for even fairly large programs in a few seconds. As expected, the pessimistic analysis is faster than the optimistic one, since it does not need to iterate to a fixpoint.

To evaluate precision and scalability, we compared our analysis results to dynamic call graphs that we obtained by manually exercising instrumented versions of our subject programs. The optimistic analysis achieves high precision ($\geq 66\%$) and very high recall ($\geq 85\%$) with respect to these dynamic call graphs, but what is perhaps surprising is that the pessimistic analysis does just as well.

This suggests that in many cases the pessimistic analysis may be preferable: not only is it faster, but it also clearly indicates which call sites it cannot reason about precisely, whereas the optimistic analysis gives a result for every call site that may sometimes be very imprecise.

Finally, we evaluate several possible client applications of our algorithms: we show that the call graphs they generate are much more complete than what current IDEs offer and could be used to significantly improve the “Jump to Declaration” feature. Moreover, our call graphs also facilitate the implementation of bug finding and smell detection tools that check for common programming mistakes.
6. **Approximate Call Graphs for JavaScript IDE Services**

![Figure 6.1: Test page before (a), and after (b) applying the plugin](image-url)

**Contributions** To the best of our knowledge, this paper presents the first static analysis capable of computing useful call graphs for large JavaScript applications. Specifically, we make the following contributions:

- We propose two variants of a field-based flow analysis for JavaScript that only tracks function objects and ignores dynamic property reads and writes.
- We show that both scale to large, real-world programs.
- Comparing against dynamic call graphs, we find that the analyses, while in principle unsound, produce accurate call graphs in practice.
- We demonstrate several client applications to show the usefulness of our approach.

The remainder of the paper is organized as follows. Section 6.2 motivates our techniques on a simple but realistic JavaScript program. Section 6.3 explains our analyses in detail, and Section 6.4 evaluates them on real-world applications. Section 6.5 surveys related work, and Section 6.6 concludes.

### 6.2 Motivating Example

As an example to introduce our techniques, we present a simple plugin to the popular jQuery framework that can be used to highlight alternating rows of HTML tables for easier on-screen reading as shown in Fig. 6.1. We discuss some challenges for call graph construction illustrated by this example, and explain how our analysis handles them.

#### 6.2.1 Example Walkthrough

The jQuery framework provides a wide variety of functionality to simplify cross-browser application development. Most of its features are exposed through the global `jQuery` function, which can be used to register event handlers, parse snippets of HTML, or perform CSS queries over the DOM. The result of such a query is a special jQuery result object, which provides array-like access to the result elements through numerical indices and offers many utility methods, some of them defined by jQuery itself, and others defined by plugins.
6.2. Motivating Example

Our simplified version of jQuery, shown in Fig. 6.2, implements a jQuery function as well. Following a common pattern, it is first defined as a local function within a surrounding closure (lines 2–9), and later stored in a global variable to make it accessible to client code (line 25). Our jQuery function only provides a very simple form of querying: when passed a string argument, it finds all DOM elements with this tag name (line 4), stores them into the result object, sets its length property to indicate how many elements were found, and returns it. For instance, jQuery('tbody') returns all table body elements in the document.

The result object itself is created on line 3 using the built-in function Object.create, which takes as its argument an object \( p \) and returns a new object \( o \) that has \( p \) as its prototype. In this case, the prototype object will be jQuery.fn, which is defined on line 11. Thus, any property defined on jQuery.fn is available on all jQuery result objects via JavaScript’s prototype-based inheritance mechanism.

Initially, the jQuery.fn object contains a single property: a method extend that adds all property-value pairs of its argument object \( \text{obj} \) to jQuery.fn. This is done through a for-in loop (lines 13–14) that iterates over all properties \( p \) of \( \text{obj} \), and uses dynamic property reads and writes to copy the value of property \( p \) on \( \text{obj} \) into a property of the same name on jQuery.fn. If no such property exists yet, it will be created; otherwise, its previous value will be overwritten.

On line 18 the extend method is used to add a method each to jQuery.fn, which iterates over all elements contained in a result object and invokes the given callback function \( \text{cb} \) on it, passing both the element and its index as arguments.

The plugin, shown in Fig. 6.3, uses the each method, passing it a callback that in turn iterates over all the children of every result element, and sets the background color
6. Approximate Call Graphs for JavaScript IDE Services

```javascript
(function($) {
$.fn.highlightAlt = function(c) {
  this.each(function(elt) {
    for(var i=1; i<elt.children.length; i+=2)
    elt.children[i].style.backgroundColor = c;
  });
};

window.highlightAltRows = function() {
$('tbody').highlightAlt('#A9D0F5');
};
}(jQuery));
```

Figure 6.3: A jQuery plugin to highlight alternating children of DOM elements

of every second element to a given color `c` (line 31). This functionality is exposed as a method `highlightAlt` added to the `jQuery.fn` object, and hence available on every jQuery result object. The plugin also defines a global function `highlightAltRows` that clients can invoke to apply highlighting to all tables in the document: it uses the `jQuery` function to find all table bodies, and then invokes `highlightAlt` on each of them. Notice that a closure is used to make the global `jQuery` variable available as a local variable `$`.

Our example illustrates several important features of JavaScript: variables have no static types and may, in general, hold values of different types over the course of program execution. Objects in JavaScript do not have a fixed set of properties; instead, properties can be created simply by assigning to them (e.g., the plugin adds a method `highlightAlt` to `jQuery.fn`), and can even be deleted (not shown in the example). Functions are first-class objects that can be passed as arguments (as with the `each` function), stored in object properties to serve as methods, and even have properties themselves. Finally, dynamic property reads and writes allow accessing properties by computed names.

6.2.2 Challenges for Call Graph Construction

As discussed in Section 6.1, call graphs are widely useful in IDEs, for example to implement “Jump to Declaration” or to perform lightweight analysis tasks. Unfortunately, neither standard coarse approaches nor more precise flow analyses work well for building JavaScript call graphs, as we shall explain using our running example.

Java IDEs take a type-based approach to call-graph construction [19]: the possible targets of a method call are simply those admitted by the program’s class hierarchy. Since variables and properties are not statically typed in JavaScript, type-based call graph construction algorithms are not immediately applicable. While prototype objects are superficially similar to Java classes, properties can be dynamically added or overwritten. For instance, the `jQuery.fn` object in our example starts out with only one property (`extend`) to which two others (`each` and `highlightAlt`) are later added, defeating any simple static type inference. Type inference algorithms for JavaScript that can handle such complications have been proposed [41, 46], but do not yet scale to real-world programs.

An very naïve way to construct call graphs would be to use name matching, and resolve a call `e.f(...)` to all functions named `f`. This approach, which is used by Eclipse JSDT, fails when functions are passed as parameters or stored in properties with a different name, like the `extend` function on line [12] Consequently, JSDT is unable to resolve any of the
call sites in our example. Other IDEs employ more sophisticated techniques, but we do not know of any current IDE that can handle callbacks and discover targets for the call on line 21.

A more conservative approach suggesting any function with the right number of parameters as a call target would likely be too imprecise in practice, yet still fails to be sound, since JavaScript allows arity mismatching: the call on line 21 passes two parameters, while the callback only declares one.

A flow analysis, like an Andersen-style pointer analysis [80] or an inter-procedural data flow analysis [46], can avoid these issues. Such analyses work by tracing the flow of abstract values through abstract memory locations based on the relevant program statements (primarily assignments and function calls). A call graph is then derived by determining which function values flow to each invoked expression.

However, building a precise flow analysis that scales to large JavaScript programs is an unsolved challenge. In the example of Fig. 6.2, the flow of functions to invocations is non-trivial, due to the use of the `extend` function to modify the `jQuery.fn` object. Precise modeling of dynamic property accesses like those in `extend` and other complex constructs is required to obtain a useful flow analysis result, but this precise modeling can compromise scalability; see [80] for a detailed discussion. In particular, we know of no JavaScript flow analysis that can analyze real-world jQuery-based applications.

6.2.3 Our Approach

In this paper, we show that a simple flow analysis suffices to construct approximate call graphs that are, in practice, sufficiently accurate for applications such as IDE services. Our analysis only tracks the flow of function values, unlike most previous flow analyses, which track the flow of all objects. Ignoring general object flow implies that for a property access `e.f`, the analysis cannot reason about which particular (abstract) object’s `f` property is accessed. Instead, a field-based approach is employed, in which `e.f` is modeled as accessing a single global location `f`, ignoring the base expression `e`.

Our analysis uses a standard flow graph capturing assignments of functions into variables and properties, and of one variable into another. For instance, the function declaration on line 2 adds a flow graph edge from the declared function to the local variable `jQuery`, while the assignment on line 25 adds an edge from that variable to the abstract location `Prop(jQuery)` representing all properties named `jQuery`. The function call on line 38, in turn, establishes a flow from `Prop(jQuery)` into the parameter `$`, leading the analysis to conclude that the call on line 36 may indeed invoke the `jQuery` function. Details of how to construct the flow graph and how to extract a call graph from it are presented in the next section.

At first glance, dynamic property accesses present a formidable obstacle to this approach: for a dynamic property access `e[p]`, the analysis cannot reason about which names `p` can evaluate to, since string values are not tracked. A conservative approximation would treat such accesses as possibly reading or writing any possible property, leading to hopelessly imprecise analysis results. However, we observe that dynamic property accesses in practice often occur as correlated accesses [80], where the read and the write refer to the same property, as on line 14 in our example. A field-based analysis can safely

---

1The analysis in [80] could only analyze a manually rewritten version of jQuery with handling of certain JavaScript features disabled.
ignore correlated accesses, since like-named properties are merged anyway. Our analysis
goes further and simply ignores all dynamic property accesses.
This choice compromises soundness, as seen in this example (inspired by code in
jQuery):

```javascript
arr = ["Width","Height"];
for (var i=0; i<arr.length; ++i)
  $.fn["outer"+arr[i]] = function() { ... };
$.fn.outerWidth();
```

The dynamic property write inside the loop corresponds to two static property writes to
$.fn.outerWidth and $.fn.outerHeight, which the analysis ignores; hence it is unable
to resolve the call to outerWidth.
But, as we shall show in our evaluation (Section 6.4), such cases have little effect
on soundness in practice. Furthermore, unlike more precise flow analyses, our approach
scales easily to large programs, which makes it well suited for use in an IDE, where a
small degree of unsoundness can be tolerated.

6.3 Analysis Formulation

We now present the details of our call graph construction algorithm. We first explain
the intraprocedural parts of the analysis, and then present two contrasting approaches to
handling interprocedural flows, one pessimistic and one optimistic.

6.3.1 Intraprocedural Flow

Our algorithm operates over a flow graph, a representation of the possible data flow induced
by program statements. The vertices of the flow graph represent functions, variables and
properties, while the edges represent assignments. To emphasize the suitability of our
techniques for an IDE, we show how to construct the flow graph directly from an AST,
as is done in our implementation.

Abstracting from a concrete AST representation, we write $\Pi$ for the set of all AST
positions, and use the notation $t^\pi$ to mean a program element $t$ (such as an expression,
a function declaration, or a variable declaration) at position $\pi \in \Pi$. We assume a lookup
function $\lambda$ for local variables such that $\lambda(\pi, x)$ returns the
position of the local variable or parameter declaration (if any) that $x$ binds to at position
$\pi$. For any position $\pi$, $\phi(\pi)$ denotes the position of the innermost enclosing function
(excluding $\pi$ itself).

There are four basic types of vertices:

$$
V \::= \begin{cases} 
\text{Exp}(\pi) & \text{value of expression at } \pi \\
\text{Var}(\pi) & \text{variable declared at } \pi \\
\text{Prop}(f) & \text{property of name } f \\
\text{Fun}(\pi) & \text{function declaration/expression at } \pi 
\end{cases}
$$

We define a function $V$ that maps expressions to corresponding flow graph vertices:

$$
V(t^\pi) = \begin{cases} 
\text{Var}(\pi') & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = \pi' \\
\text{Prop}(x) & \text{if } t \equiv x \text{ and } \lambda(\pi, x) \text{ undefined} \\
\text{Prop}(f) & \text{if } t \equiv e.f \\
\text{Exp}(\pi) & \text{otherwise}
\end{cases}
$$
6.3. Analysis Formulation

<table>
<thead>
<tr>
<th>node at ( \pi )</th>
<th>edges added when visiting ( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R1) ( l = r )</td>
<td>( V(r) \rightarrow V(l), V(r) \rightarrow \text{Exp}(\pi) )</td>
</tr>
<tr>
<td>(R2) ( l \mid</td>
<td>\ r )</td>
</tr>
<tr>
<td>(R3) ( t ? l : r )</td>
<td>( V(l) \rightarrow \text{Exp}(\pi), V(r) \rightarrow \text{Exp}(\pi) )</td>
</tr>
<tr>
<td>(R4) ( l &amp;&amp; r )</td>
<td>( V(r) \rightarrow \text{Exp}(\pi) )</td>
</tr>
<tr>
<td>(R5) { f: ( \epsilon }}</td>
<td>( V(\epsilon) \rightarrow \text{Prop}(f) )</td>
</tr>
<tr>
<td>(R6) function expression</td>
<td>( \text{Fun}(\pi) \rightarrow \text{Exp}(\pi) ), if it has a name: ( \text{Fun}(\pi) \rightarrow \text{Var}(\pi) )</td>
</tr>
<tr>
<td>(R7) function declaration</td>
<td>( \text{Fun}(\pi) \rightarrow \text{Var}(\pi) )</td>
</tr>
</tbody>
</table>

Figure 6.4: Intraprocedural flow graph edges generated for AST nodes

![Intraprocedural flow graph edges generated for AST nodes](image)

Figure 6.5: Partial flow graph for Fig. 6.2 and 6.3. Solid edges are added by the rules of Fig. 6.4 dotted edges by the rules of Fig. 6.6 and dashed edges by Alg. 2.

To build the flow graph, we traverse the AST and add edges as specified by the rules in Fig. 6.4. For our example, by rule (R7) the declaration of jQuery on line 2 yields an edge \( \text{Fun}(2) \rightarrow \text{Var}(\text{jQuery}) \), where we use line numbers as positions and refer to local variables by name for readability. Likewise, the function expression on line 12 yields two edges \( \text{Fun}(12) \rightarrow \text{Var}(\text{ext}) \) and \( \text{Fun}(12) \rightarrow \text{Exp}(12) \) by (R6). Some of the other edges generated for our example are shown as solid arrows in the partial flow graph in Fig. 6.5.

6.3.2 Interprocedural Flow

To handle interprocedural flow, the set of vertices needs to be extended as follows:

\[
V ::= \ldots
|\text{Callee}(\pi) \quad \text{callee of call at } \pi
|\text{Arg}(\pi, i) \quad \text{i\text{th argument of call at } } \pi
|\text{Parm}(\pi, i) \quad \text{i\text{th parameter of function at } } \pi
|\text{Ret}(\pi) \quad \text{return value of function at } \pi
|\text{Res}(\pi) \quad \text{result of call at } \pi
\]

(R4) is somewhat subtle: in JavaScript, the result of \( l \&\& r \) can only be \( l \) if \( l \) evaluates to a false value, but in this case it is not a function, and thus does not have to be tracked.
The function $V$ mapping expressions to vertices is likewise extended: if $\lambda(\pi, x)$ is the $i$th parameter of the function declared at $\pi'$, then $V(x^\pi) = \text{Parm}(\pi', i)$, and $V(\text{this}^\pi) = \text{Parm}(\phi(\pi), 0)$, i.e., \textbf{this} is considered to be the 0th parameter. Rules for connecting $\text{Arg}$ and $\text{Ret}$ vertices with $\text{Exp}$ vertices are given in Fig. 6.6.

Returning to our example, the function call on lines 27–38, yields, by rule (R8), an edge $\text{Prop}(\text{jQuery}) \rightarrow \text{Arg}(38, 1)$. This edge, and some of the other edges that are generated by the rules of Fig. 6.6 are shown as dotted arrows in Fig. 6.5.

We now introduce two approaches for connecting $\text{Parm}$ to $\text{Arg}$ and $\text{Ret}$ to $\text{Res}$ vertices to track interprocedural flow.

**Algorithm 1**

**Pessimistic Call Graph Construction**

**Output:** call graph $C$, escaping functions $E$, unresolved call sites $U$

1: $C_i := \{(\pi, \pi') | t^\pi \text{ is a one-shot call to a function } f^\pi\}$
2: $E_i := \{\pi' | \neg \exists \pi. ((\pi, \pi') \in C_i\}$
3: $U_i := \{\pi | \neg \exists \pi'. (\pi, \pi') \in C_i\}$
4: $G := \emptyset$
5: Add Interprocedural Edges $(G, C_i, E_i, U_i)$
6: add edges from Fig. 6.4 and 6.6
7: $C := \{(\pi, \pi') | \text{Fun}(\pi') \sim_G \text{Callee}(\pi)\}$
8: $E := \{\pi | \text{Fun}(\pi) \sim_G \text{Unknown}\}$
9: $U := \{\pi | \text{Unknown} \sim_G \text{Callee}(\pi)\}$

### 6.3.3 Pessimistic Approach

The pessimistic call graph construction algorithm (Alg. 1) only tracks interprocedural flow in the important special case of \textit{one-shot calls}, i.e., calls of the form

$$\text{(function}(\pi) \{ \ldots \})(\pi)$$

where an anonymous function (the \textit{one-shot closure}) is directly applied to some arguments. In all other cases, interprocedural flow is modeled using a special \textbf{Unknown} vertex.

We start call graph construction from an initial call graph $C_i$ that only contains edges from one-shot calls to one-shot closures. All other functions are considered \textit{escaping functions} (set $E_i$), and all other call \textit{unresolved call sites} (set $U_i$). The flow graph $G$ is initially empty.

Now we add interprocedural edges to $G$ as described in Alg. 2. \textbf{Arg} vertices are connected to $\text{Parm}$ vertices along the edges in $C_i$, and similar for $\text{Ret}$ and $\text{Res}$, thus

<table>
<thead>
<tr>
<th>(R8)</th>
<th>node at $\pi$</th>
<th>edges added when visiting $\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(\pi)$ or new $f(\pi)$</td>
<td>$V(f) \rightarrow \text{Callee}(\pi)$, $V(e_i) \rightarrow \text{Arg}(\pi, i)$, $\text{Res}(\pi) \rightarrow \text{Exp}(\pi)$</td>
<td></td>
</tr>
<tr>
<td>(R9)</td>
<td>$r.p(\pi)$</td>
<td>as (R8), plus $V(r) \rightarrow \text{Arg}(\pi, 0)$</td>
</tr>
<tr>
<td>(R10)</td>
<td>return $e$</td>
<td>$V(e) \rightarrow \text{Ret}(\phi(\pi))$</td>
</tr>
</tbody>
</table>

Figure 6.6: Flow graph edges generated for calls and returns
Algorithm 2 Add Interprocedural Edges

Input: flow graph $G$, initial call graph $C$, escaping functions $E$, unresolved call sites $U$

1. for all $(\pi, \pi') \in C$ do
2.   add edges $Arg(\pi, i) \rightarrow Parm(\pi', i)$ to $G$
3.   add edge $Ret(\pi') \rightarrow Res(\pi)$ to $G$
4. end for
5. for all $\pi \in U$ do
6.   add edges $Arg(\pi, i) \rightarrow Unknown$ to $G$
7.   add edge $Unknown \rightarrow Res(\pi)$ to $G$
8. end for
9. for all $\pi' \in E$ do
10.  add edges $Unknown \rightarrow Parm(\pi', i)$ to $G$
11.  add edge $Ret(\pi') \rightarrow Unknown$ to $G$
12. end for

modeling parameters and return values. Argument values at unresolved call sites flow into $Unknown$, and from there into every parameter of every escaping function. Conversely, the return value of every escaping function flows into $Unknown$, and from there into the result vertex of every unresolved call site.

In Fig. 6.5, this step adds the dashed edges. Note that $Arg(38, 1)$ is connected to $Parm(27, 1)$, precisely modeling the one-shot call at line 38 whereas $Arg(29, 1)$ is conservatively connected to $Unknown$, since this call site is unresolved.

Intraprocedural edges are now added as per Fig. 6.4 and 6.6.

To extract the final call graph, we need to compute the transitive closure of $G$ to determine all function vertices $Fun(\pi)$ from which a call site $\pi'$ is reachable. However, if we consider flows through $Unknown$, the resulting call graph will be very imprecise. Instead, we want to produce a call graph that gives reasonably precise call targets for many call sites, and marks sites for which no precise information is available.

Writing $\Rightarrow_G$ for the transitive closure of $G$, and $\Rightarrow_G^{opt}$ for the optimistic transitive closure which does not consider paths through $Unknown$, we define the call graph $C$, the set $E$ of escaping functions, and the set $U$ of unresolved call sites: a call may invoke any function that directly flows into its callee vertex without going through $Unknown$; if $Unknown$ flows into a site, then that site is unresolved and the information in $C$ may not be complete; and if a function flows into $Unknown$, it may be invoked at call sites not mentioned in $C$.

In the partial flow graph in Fig. 6.5, we can see that $Fun(12) \Rightarrow_G^{opt} Callee(18)$, so the call at line 18 may invoke the function at line 12 and likewise $Fun(2) \Rightarrow_G^{opt} Callee(36)$. However, $Fun(29) \not\Rightarrow_G Callee(21)$, and since there are no other flows into $Callee(21)$, the pessimistic call graph does not provide a call target for this call.

6.3.4 Optimistic Approach

The pessimistic approach produces a call graph triple $(C, E, U)$ from an initial triple $(C_i, E_i, U_i)$, which could be done repeatedly. This is what the optimistic approach does, but instead of starting from a conservative triple that considers all calls unresolved and
all functions escaped unless they are one-shot, the optimistic approach starts with the empty triple \((\emptyset, \emptyset, \emptyset)\). The flow graph is built using the same rules as for the pessimistic approach and a new triple is extracted in the same way, but then the whole procedure is repeated until a fixpoint is reached.

For our example, this leads to a more complete call graph; in particular, the optimistic approach can show that the call on line \texttt{21} may invoke the function passed on line \texttt{29}.

6.3.5 Discussion

One would expect the pessimistic approach to be more efficient but less precise than the optimistic approach, and past work on call graph construction for other languages supports this conclusion \cite{35}. As we will show in our evaluation, however, the loss in precision is actually fairly minor in practice, hence the pessimistic approach may be preferable for some applications.

Many call graph algorithms only produce call graphs for code that is deemed reachable from a given set of entry points, which can improve precision, particularly for optimistic call graph construction. We choose not to do so for two main reasons. Firstly, we want our algorithms to be usable in an IDE while developing a program; at this point, some code may not yet have been integrated with the rest of the program and hence appear to be dead, but a programmer would still expect IDE services to be available for this code.

Secondly, reasoning about reachability requires a fairly elaborate model of the JavaScript standard library and the DOM: for instance, event handlers should always be considered reachable, and reflective invocations using \texttt{call} and \texttt{apply} must also be accounted for. By analyzing all code instead, we can make do with a very coarse model that simply lists all known methods defined in the standard library and the DOM. For a standard library function such as \texttt{Array.prototype.sort}, we then simply introduce a new vertex \texttt{Builtin(Array_sort)} with an edge to \texttt{Prop(sort)}.

6.4 Evaluation

We have implemented both the pessimistic and the optimistic call graph algorithm in CoffeeScript\cite{3}, a dialect of JavaScript. In this section, we evaluate our implementation with respect to the following three evaluation criteria:

(EC1) How scalable are our techniques?

(EC2) How accurate are the computed call graphs?

(EC3) Are our techniques suitable for building IDE services?

To evaluate these criteria, we run both our algorithms on ten real-world subject programs and measure their performance. To measure accuracy, we compare the resulting static call graphs against dynamic call graphs obtained by manually exercising the programs. Finally, we informally compare our analyses with existing IDEs, and report on experiments with two client analyses implemented on top of our call graphs.\textsuperscript{4}

\footnotesize \textsuperscript{3}http://coffeescript.org/
\footnotesize \textsuperscript{4}Our experimental data is available online at http://tinyurl.com/jscallgraphs
6.4. Evaluation

Table 6.1: Subject programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Underlying Framework</th>
<th>LOC</th>
<th>Num. of Functions</th>
<th>Num. of Calls</th>
<th>Dyn. CG</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3dmodel</td>
<td>none</td>
<td>4880</td>
<td>29</td>
<td>109</td>
<td>55.17%</td>
<td></td>
</tr>
<tr>
<td>beslimed</td>
<td>MooTools</td>
<td>4750</td>
<td>703</td>
<td>2017</td>
<td>86.05%</td>
<td></td>
</tr>
<tr>
<td>coolclock</td>
<td>jQuery</td>
<td>6899</td>
<td>548</td>
<td>1747</td>
<td>81.25%</td>
<td></td>
</tr>
<tr>
<td>flotr</td>
<td>Prototype</td>
<td>4946</td>
<td>743</td>
<td>2671</td>
<td>68.98%</td>
<td></td>
</tr>
<tr>
<td>fullcalendar</td>
<td>jQuery</td>
<td>12265</td>
<td>1089</td>
<td>4083</td>
<td>70.83%</td>
<td></td>
</tr>
<tr>
<td>htmledit</td>
<td>jQuery</td>
<td>3606</td>
<td>389</td>
<td>1253</td>
<td>62.00%</td>
<td></td>
</tr>
<tr>
<td>markitup</td>
<td>jQuery</td>
<td>6471</td>
<td>557</td>
<td>1849</td>
<td>71.43%</td>
<td></td>
</tr>
<tr>
<td>pacman</td>
<td>none</td>
<td>3513</td>
<td>152</td>
<td>485</td>
<td>79.61%</td>
<td></td>
</tr>
<tr>
<td>pdfjs</td>
<td>none</td>
<td>31694</td>
<td>965</td>
<td>3570</td>
<td>67.77%</td>
<td></td>
</tr>
<tr>
<td>pong</td>
<td>jQuery</td>
<td>3646</td>
<td>375</td>
<td>1324</td>
<td>75.00%</td>
<td></td>
</tr>
</tbody>
</table>

6.4.1 Subject Programs

Table 6.1 lists our subjects, which are ten medium to large browser-based JavaScript applications covering a number of different domains, including games (beslimed, pacman, pong), visualizations (3dmodel, coolclock), editors (htmledit, markitup), a presentation library (flotr), a calendar app (fullcalendar), and a PDF viewer (pdfjs). As shown in the table, all but three of them rely on frameworks; these frameworks are the three most widely-used ones according to a recent survey [90], which found that 56% of all surveyed websites used jQuery, 5% used MooTools, and 4% used Prototype.

In many cases, the framework libraries included in the subject programs were in mini-

fied form. To aid debugging, we replaced these by their unminified development versions, which also more closely matches the development setting in which we envision our techniques to be used. Since minifiers typically do not rename properties, however, our analyses should not be significantly less precise for minified code.

For each program, we list three size measures: the number of non-blank, non-comment lines of code as determined by the cloc utility, as well as the number of functions and of call sites. The coverage number in the last column will be explained below. For the framework-based subjects, framework code contributes between 66% and 94% of code size.

6.4.2 Scalability (EC1)

To evaluate scalability, we measured the time it takes to build call graphs for our subject programs using both of our algorithms. As JavaScript’s built-in time measurement functions turned out to be unreliable, we used the standard UNIX time command, measuring user time. This includes both time for parsing and for the analysis, so we separately measured the time it takes just to parse every program.

The results of these measurements are given in Fig. 6.7. All experiments were performed on a Lenovo ThinkPad W520 with an Intel Core i7-2720QM CPU and 8GB RAM, using version 3.1.8.22 of the V8 JavaScript engine running on Linux 3.0.0-24 (64-bit version). Timings are averaged over ten runs, with error bars indicating standard deviation.

Both analyses scale very well, with even the largest program analyzed in less than 18 seconds using the optimistic, and less than nine seconds using the pessimistic approach (including about three seconds of parsing time in both cases). The pessimistic analysis in
6. Approximate Call Graphs for JavaScript IDE Services

Figure 6.7: Time measurements for parsing and analysis; averaged over ten runs, error bars indicate standard deviation

Figure 6.8: Precision and recall measurements for optimistic and pessimistic call graphs compared to dynamic call graphs

particular already seems fast enough for use in an IDE, where an AST would already be available.

6.4.3 Call Graph Accuracy (EC2)

Measuring the accuracy of our call graphs is not easy, since there is no existing analysis that can handle all our subject programs against which to compare our results. Instead we compare against dynamic call graphs and measure precision and recall with respect to dynamically observed call targets.

To obtain dynamic call graphs, we instrumented our subject programs to record the observed call targets for every call that is encountered at runtime, and manually exercised these instrumented versions. Additionally, we measured the function coverage achieved this way, i.e., the percentage of non-framework functions that were executed while recording the call graphs, which is shown in the last column of Table 6.1. In all cases but one, coverage is above 60%, indicating that the dynamic call graphs are based on a reasonable portion of the code and hence likely to be fairly complete. We manually investigated the low coverage on 3dmodel and found that most of the uncovered code does in fact seem to be dead.
Next, we used our two analyses to generate call graphs for all our subject programs, and computed precision and recall for every call site that is covered by the dynamic call graph. Writing $D$ for the set of targets of a given call site in the dynamic call graph, and $S$ for the set of targets determined by the analysis, the precision is computed as $\frac{|D \cap S|}{|S|}$ (i.e., the percentage of “true” call targets among all targets), while recall is $\frac{|D \cap S|}{|D|}$ (i.e., the percentage of correctly identified true targets). Averaging over all call sites for a given program, we obtain the results in Fig. 6.8.

Both analyses achieve almost the same precision on most programs, with the pessimistic analysis performing slightly better. Only on beslimed and flotr, the two non-jQuery programs, is the difference more marked, and we only achieve a relatively modest precision of between 65% and 75%, while on the others the precision is at least 80%.

For both approaches, the main sources of imprecision are functions that are stored in properties of the same name, which a field-based analysis cannot distinguish as call targets. Additionally, the optimistic approach may resolve callback invocations imprecisely. The pessimistic approach would give up on such call sites, returning zero call targets, which accounts for its better precision measure.

Both analyses achieve very high recall: in every case, more than 80% of dynamically observed call targets are also found by the analysis, with recall above 90% for the jQuery-based programs and close to 100% for the framework-less programs. Missing call targets are due to the unsoundness of our approach with respect to dynamic property writes. These are often used in frameworks to define a group of closely related functions or to do metaprogramming, which is rare in non-framework code. On flotr, the optimistic analysis does significantly better than the pessimistic one; this seems to be due to a liberal use of callback functions, which are not handled by the pessimistic analysis.

### 6.4.4 Suitability for IDE Services (EC3)

We now evaluate the suitability of our analyses for three typical client applications.

**Jump to Declaration**  
Java IDEs typically offer a “Jump to Declaration” feature for navigating from a field or method reference to its declaration. In JavaScript, there are no method declarations as such, but several JavaScript IDEs offer a similar feature to navigate from a function call to the function (or, in general, functions) that may be invoked at this place.

Our call graph algorithms could be used to implement such a feature. The pessimistic algorithm seems to be particularly well-suited, since it gives a small set of targets for most call sites. While no call target may be available for unresolved call sites, this is arguably better than listing many spurious targets.

To test this hypothesis, we measure the percentage of call sites with a single target, excluding native functions. The results, along with the percentage of call sites with zero, two, three, and more than three targets, are given in Fig. 6.9 on all benchmarks, more than 70% of call sites have at most one target, 80% have at most two and 90% at most three targets. This suggests that the pessimistic algorithm could be useful for implementing Jump to Declaration.

The relatively large percentage of call sites without targets is due to excluding native call targets. If they are included, the pessimistic analysis is on average able to find at least
one callee for more than 95% of calls. The maximum number of non-native call targets is 20 callees for a small number of sites on beslimed; if native targets are considered, several call sites can have up to 124 callees: these are calls to `toString`, with 120 of the suggested callees being DOM methods.

We now compare our approach against three current JavaScript IDEs: Eclipse JSDT, Komodo IDE, and WebStorm.

The Eclipse JSDT plugin (we tested version 1.4.0 on Eclipse 4.2.0) provides a Jump to Declaration feature, which does not seem to handle method calls, severely limiting its practical usefulness: across all our subject programs, it can only find targets for about 130 call sites (less than 1%).

Komodo IDE (version 7.0.2) uses fairly intricate heuristics to resolve function and method calls that work well on our smaller subject programs such as 3dmodel. However, it seems unable to handle larger, framework-based programs, where its Jump to Declaration feature usually fails.

WebStorm (version 4.0.2) is closed-source, precluding examination of its implementation. It seems to maintain a representation of variable and property assignments similar to our flow graph. No transitive closure is computed, hence Jump to Declaration only jumps to the most recent definition and it may take several jumps to find the actual callee. WebStorm has built-in support for the most popular frameworks, so it can understand commonly used metaprogramming patterns that foil our analyses. However, it performs no interprocedural reasoning at all (not even for one-shot closures), so it is impossible to jump to the declaration of a callback function.

**Smell detection** As an example of a more complicated client, we implemented a simple tool for detecting global variables that are used like local variables, suggesting a missing `var` declaration. While this may not necessarily be a bug, it is considered bad practice and makes code less robust.

We check whether all functions using a global variable \( x \) definitely assign to \( x \) before reading it. Additionally, call graph information is used to check whether one function using \( x \) can call another. If so, the functions might see each others’ updates to \( x \), indicating that it may not be possible to turn \( x \) into a local variable without changing program behavior.
With pessimistic call graphs, the tool suggests 37 missing `var` declarations on our subject programs. One of these is a false positive due to unsoundness, but in all 36 other cases the global variable could indeed be turned into a local. With optimistic call graphs, there are only 24 true positives and the same false positive. Without interprocedural analysis, the number of false positives rises to nine: in all eight new cases, the global variable is a flag that is indeed modified by a callee and hence cannot be made local, highlighting the importance of interprocedural reasoning for this analysis.

**Bug finding** We implemented a tool that looks for functions that are sometimes called using `new`, but as normal functions at other times. While there are functions that can be used either way, this is often indicative of a bug.

Using pessimistic call graphs, the tool reports 14 such functions. One of these is a true positive indicating a bug in `flotr`, four are true but harmless positives in jQuery, and nine are false positives due to imprecise call graph information. Using optimistic call graphs, the number of false positives increases to 16, with no additional true positives.

### 6.4.5 Summary and Threats to Validity

Our evaluation shows that both call graph construction algorithms scale very well. Even though our current implementation is written in CoffeeScript and does not use highly optimized data structures, it is able to build call graphs for substantial applications in a few seconds. The faster pessimistic algorithm may be more suitable for IDE use, but further optimizations to the optimistic algorithm are certainly possible.

Comparing against dynamic call graphs, we found that the vast majority of dynamically observed call targets are predicted correctly by our analyses, and on average the number of spurious call targets is low. Our analyses resolve most call sites to at most one callee (up to 90% on some programs), and compute no more than three possible targets for almost all sites. The only extreme outliers are calls to `toString`, which have more than 100 callees due to our field-based approach.

An informal comparison of our analyses with existing IDEs suggests that the pessimistic analysis outperforms most of them, while the optimistic analysis can handle cases that exceed the capabilities of all surveyed tools. We also discussed two examples of analysis tools that need call graphs. While these tools could be useful for developers, they did not find many bugs on our subject programs, which seem quite mature.

Finally, we discuss threats to the validity of our evaluation.

First, our subject programs may not be representative of other JavaScript code. We only consider browser-based applications, so it is possible that our results do not carry over to other kinds of JavaScript programs. Most of our subject programs use jQuery, with only two programs using other frameworks. We have shown that our approaches perform particularly well on jQuery-based and framework-less applications, and slightly less so on other frameworks. On the other hand, recent data [90] suggests that less than 20% of websites use a framework other than jQuery, so our approach should be applicable to most real-world, browser-based JavaScript code.

Second, our accuracy measurements are relative to an incomplete dynamic call graph, not a sound static call graph. Hence the recall should be understood as an upper bound (i.e., recall on a more complete call graph could be lower), whereas precision is a lower bound (i.e., precision could be higher). Given the difficulty of scaling sound call graph
algorithms to realistic programs, dynamic call graphs are the best data we can compare ourselves against at the moment. Moreover, the relatively high function coverage of the dynamic call graphs suggests that they are representative of the entire programs.

6.5 Related Work

Existing flow analyses for JavaScript [27, 36, 37, 46, 80, 87] generally do not scale to framework-based programs. Some of them, such as Gatekeeper [36], do not reason statically about dynamic property accesses, just like our analysis. Gatekeeper recovers soundness, however, by performing additional runtime instrumentation. All these systems track general object flow and use a more precise heap model than we do.

Recently, Madsen et al. presented an analysis that side-steps the problem of analyzing complex framework code and modeling native APIs by inferring their behavior from uses in client code [59]. Our approach is scalable enough to directly analyze frameworks, and since we only track functions and do not reason about reachability, no elaborate models for native code seem to be necessary. We could, however, adopt their approach in cases where such modeling becomes important.

Wei and Ryder [91] propose a combined static-dynamic taint analysis of JavaScript programs. In their work, a number of traces is collected that contain information about method calls and object creation. This information is used to assist a static taint analysis with the construction of a call graph that includes code that is executed as the result of calls to `eval`, and excludes code in uncovered branches. In addition, the number of arguments supplied to methods calls is captured and used to counter some of the loss of precision due to function variadicity, by creating multiple distinct nodes in the call graph for certain methods. Like ours, their analysis is unsound, but it is likely to be less scalable than ours because of its reliance on a traditional static pointer analysis. An in-depth comparison of cost and accuracy of the two approaches is future work.

Agesen et al. presented a number of type inference techniques [1–3] for Self, a language with many similarities to JavaScript. They essentially compute highly context-sensitive flow graphs (from which call graphs could be extracted) to statically prove the absence of “message not understood” errors, where a method is not found on the receiver object or its prototypes. Our technique cannot do such reasoning, since it does not track the flow of most objects. Tracking general object flow for JavaScript leads to scalability and precision issues due to heavy usage of reflective idioms that seem not to be as frequently used in Self.

Grove and Chambers [35] present a general framework for call-graph construction algorithms. Our analysis does not fit directly in their framework since they do not discuss prototype-based languages, but roughly speaking, our analysis can be viewed as a variant of 0-CFA [77] where (1) only function values are tracked, (2) field accesses are treated as accesses to correspondingly-named global variables, and (3) all code is assumed to be reachable. Our key contribution is in showing that such an analysis works well for JavaScript in practice. Previous work has studied the effectiveness of field-based flow analysis for C [43] and Java [57, 79]. They exploit static type information to distinguish identically named fields of different struct/class types, which is impossible in JavaScript.
6.6 Conclusions

We have presented a fast, practical flow analysis-based approach to call graph construction for JavaScript. Our analysis (i) is field-based, i.e., identically named properties of different objects are not distinguished; (ii) only tracks function values, ignoring the flow of other objects; and (iii) ignores dynamic property reads and writes. We have proposed two variants of this analysis: a pessimistic variant that makes conservative assumptions about interprocedural flow, and an optimistic variant that iteratively builds an interprocedural flow graph.

Both analyses scale extremely well and can handle far larger programs than any other static analysis for JavaScript that we are aware of. While unsound in theory, they produce fairly complete call graphs in practice. These properties make our approach well-suited for use in an IDE.

In such a setting, it would be wasteful to build a call graph from scratch every time it is needed, since large parts of the program typically remain unchanged. Instead, flow graphs could be precomputed and cached on a per-file basis, and then combined into a graph for the whole program when needed.

As future work, we plan to apply our approach in other settings besides IDEs, such as taint analysis [38]. Here, soundness is much more important, so we need to handle dynamic property accesses. Conservatively treating them as potentially accessing all properties will in general result in too much imprecision, so some form of string analysis for reasoning about property names is likely needed. Introducing this and other features (such as tracking of non-function objects) into our analysis while still keeping it scalable is an interesting challenge, which could provide valuable insights into the cost and benefit of different analysis features for JavaScript.
Chapter 7

Semi-Automatic Rename Refactoring for JavaScript

This chapter is a copy of an OOPSLA’13 article [24] by Asger Feldthaus and Anders Møller.

Abstract

Modern IDEs support automated refactoring for many programming languages, but support for JavaScript is still primitive. To perform renaming, which is one of the fundamental refactorings, there is often no practical alternative to simple syntactic search-and-replace. Although more sophisticated alternatives have been developed, they are limited by whole-program assumptions and poor scalability.

We propose a technique for semi-automatic refactoring for JavaScript, with a focus on renaming. Unlike traditional refactoring algorithms, semi-automatic refactoring works by a combination of static analysis and interaction with the programmer. With this pragmatic approach, we can provide scalable and effective refactoring support for real-world code, including libraries and incomplete applications. Through a series of experiments that estimate how much manual effort our technique demands from the programmer, we show that our approach is a useful improvement compared to search-and-replace tools.

7.1 Introduction

Refactoring is the process of transforming the source code of a program to enhance its internal structure while preserving its external behavior. Many kinds of refactoring are used in modern software development, for example, renaming of fields or methods, extracting blocks of code into separate methods, or moving code from one package to another [29]. Refactoring tools in IDEs assist the programmer in performing the necessary source code transformations and checking that the program behavior is preserved. However, the most powerful existing techniques that provide such automated support for refactoring have been developed for statically typed programming languages, such as Java and C#. As those techniques critically depend on static information about class hierarchies, packages, and types of fields and methods, they cannot easily be adapted to dynamically typed languages. In JavaScript, for example, the object properties change at runtime, and classes and packages are at best mimicked by meta-programming in libraries, which are difficult
to reason about statically. The refactoring techniques that exist for dynamically typed languages, in particular JavaScript, remain primitive or impractical:

(1) One approach, introduced in the Smalltalk Refactoring Browser [70], is to use runtime instrumentation to detect the consequences of tentative program transformations. This requires extensive test suites to ensure that the program behavior is preserved, and we are not aware of this approach being used in practice for JavaScript.

(2) In previous work [27], we showed that static points-to information can be used in lieu of static types as the foundation for automated refactoring for JavaScript. Although this approach works well in many situations, we have observed some practical limitations. Pointer analysis is hard to scale to large JavaScript programs without sacrificing the necessary precision. Despite much effort, no existing pointer analysis can handle typical JavaScript programs that use libraries. Moreover, the technique only works for complete programs, which means it cannot reliably be applied to library code, except if a comprehensive test suite is available, or to application code that is under development where parts of the code have not yet been written.

(3) If we consider one of the most common kinds of refactoring—renaming an object property—the working programmer’s best tool is still search-and-replace. This can be based on the plain text or the abstract syntax tree of the program source code, in either case without any consideration toward its semantics. If the programmer wishes to rename an object property, he may choose to rename all occurrences of the selected symbol in one fell swoop, or pick one-by-one the occurrences he wishes to rename. The former is prone to errors since unrelated objects may have properties with the same name, while the latter can be rather tedious if there are many occurrences.

Several IDEs exist for JavaScript programming, including Eclipse JSDT, Eclipse Orion, NetBeans, Visual Studio for Web, Komodo IDE, Cloud9, and WebStorm. Among all these, we find only one—WebStorm— that supports rename refactoring for JavaScript. While this tool is quite effective for renaming local variables, it seems to fall back to search-and-replace when renaming object properties. Based on brief experimentation, it appears that it always replaces all occurrences of the given object property name. WebStorm is closed source and its algorithms are not publicly documented, so we cannot verify how the tool works.

In this work, we propose a semi-automatic technique for renaming object properties in JavaScript programs. Renaming is among the most common kinds of refactoring [62, 85]. Making better tool support for renaming than search-and-replace requires information about how object property access operations in the program code are related to each other. Such information is also fundamental for other refactorings, for example, property encapsulation [27]. Our approach may be seen as a middle ground between fully automated refactoring and tedious one-by-one search-and-replace. When the programmer decides that some occurrence of an object property \( x \) in the program needs to be renamed to \( y \), we use a lightweight static analysis to divide all occurrences of \( x \) into groups. Two occurrences of \( x \) are placed in the same group if the analysis infers that they are related in the sense that they must either both be renamed or neither be renamed. The refactoring tool then asks the programmer whether each group should be renamed entirely or left unchanged. In this way, the programmer gets one question for each group, and each question only mentions one occurrence of \( x \). Compared to traditional search-and-replace, this approach

\footnote{http://www.jetbrains.com/webstorm/}
requires less effort by the programmer since we avoid asking questions where the answer can be inferred from previous answers. Compared to the fully automated approach from our previous work, we circumvent the computationally expensive pointer analysis and the whole-program assumption.

For an incomplete program—for example, a library or an application under development—ultimately only the programmer knows whether two object property operations are related or not, given that no formal specification exists of the desired program behavior. Still, we can define an informal notion of soundness for our static analysis: if the heuristics employed by the analysis group together two occurrences of $x$, then it should not be the case that the programmer wants one of them to be renamed but not the other, since he only gets a question for one of them. Our static analysis therefore aims to compute a lower approximation of relatedness. The converse case, dividing the occurrences of $x$ into too many groups, is tolerable although undesirable, since the programmer must then answer more questions to complete the refactoring. For complete programs, which have a well-defined meaning, it is reasonable to assume that two occurrences of $x$ are related if they refer to the same object property in some execution of the program. This makes our analysis akin to alias analysis, but designed to work also on incomplete programs. The analysis is expressed as a set of rules reminiscent of a type system. We evaluate how it works on incomplete programs by measuring the stability of the analysis behavior when source code is added or removed from the codebase. The analysis is theoretically unsound, but our experiments indicate that it is sound in practice.

Some refactoring tools follow a tradition of performing safety checks to ensure that the program behavior will remain unchanged when performing refactorings. If a precondition is violated, such tools will abort or issue a warning to the programmer. For example, a common precondition for renaming a field in a Java program is that it does not cause a name clash. These safety checks are usually not sound in practice. Java refactoring tools typically ignore the possibility of reflection, while refactoring for JavaScript may give some leeway regarding use of the `eval` function. In the opposite end of the spectrum, other refactoring tools assume that programmer knows what he is doing, so no safety checking is performed. We follow the latter approach. The foremost rationale for this decision is that proper precondition checking requires static analysis that scales to large real-world codebases, has high precision, and works on incomplete programs, which is beyond the capabilities of existing JavaScript analysis techniques. We argue that this decision is not as detrimental to safety for semi-automatic refactoring as it would be for a fully automatic tool: the programmer will observe a number of proposed changes in the code while interacting with the tool, so he is more likely to detect problematic situations.

Reflective property access is common in JavaScript. Such expressions access an object property whose name is computed at runtime. Reflective property accesses cannot be renamed directly, since the source code does not mention the property name explicitly, and it lies beyond the scope of our proposed algorithm to update the relevant string operations that compute the property name. Our algorithm will only rename identifier tokens in the source code, and the user is expected to know this. Apart from this assumption, the internals of our proposed algorithm do not concern the programmer when performing refactorings.

Our contributions can then be summarized as follows:
We present a novel approach to semi-automatic refactoring, based on an algorithm for finding groups of related identifier tokens. A key component of this algorithm is a type inference technique reminiscent of alias analysis.

We present experiments that evaluate the practical usefulness of the approach. The results show that (1) it asks significantly fewer questions than search-and-replace, (2) it scales to large real-world codebases, (3) it does not require the whole program to be useful, and (4) although unsound in theory, the unsoundness rarely manifests in practice.

In Section 7.2 we motivate our technique using a real-world code snippet. In Section 7.3 we present the algorithm and demonstrate it on a few examples, and in Section 7.4 we present research questions and results of the experimental evaluation. Related work is discussed in Section 7.5, and our conclusions and suggestions for further work are given in Section 7.6.

### 7.2 Motivating Example

Renaming a local variable is trivial in most cases, as the scoping rules in JavaScript let us find all tokens that refer to the variable. However, renaming object properties is nontrivial. In this section, we will demonstrate some of the challenges involved in renaming object properties.

Consider the code fragment in Figure 7.1, taken from the popular jQuery library.

2

We shall intentionally withhold information about the rest of jQuery for now, and see what we can learn about the code without providing any context. The code fragment contains a top-level statement that creates a new object with six properties, three of which are initialized to function expressions. The new object is then stored in the `prototype` property of the `jQuery.Event` object. The `prototype` property has a special meaning in JavaScript: informally, objects created using a constructor call `new jQuery.Event` will inherit the properties of `jQuery.Event.prototype`.

We will now discuss how an editor with our refactoring technique reacts to various renaming requests from a programmer, assuming the code in Figure 7.1 is the only code open in the editor. Although refactoring a fragment of jQuery is an unusual scenario, the code in Figure 7.1 is representative of some of the issues we generally encounter when refactoring incomplete programs.

Consider the three `preventDefault` tokens on lines 2, 6, and 7. Suppose the programmer has decided to rename the token on line 2. It is not immediately evident whether the tokens on line 6 and 7 should be renamed as well. The IDE therefore asks the programmer whether the token on line 6 should be renamed also. However, regardless of the answer, the IDE will not ask a second time whether the token on line 7 should be renamed. Clearly, if one of these tokens were to be renamed, the other must also be renamed. We call such tokens related. In this particular case, our technique will determine that they are related because they both occur as the name of a property accessed on the `e` variable. In short, we say they access the same property on `e`.

The `originalEvent` tokens on lines 4 and 14 access a property on `this` but in different functions. Technically, any function can be invoked with an arbitrary `this` argument, and

---

2 http://jquery.com/
7.2. Motivating Example

```javascript
jQuery.Event.prototype = {
    preventDefault: function() {
        this.isDefaultPrevented = returnTrue;
        var e = this.originalEvent;
        if ( !e ) { return; }
        if ( e.preventDefault ) {
            e.preventDefault();
        } else {
            e.returnValue = false;
        }
    },
    stopPropagation: function() {
        this.isPropagationStopped = returnTrue;
        var e = this.originalEvent;
        if ( !e ) { return; }
        if ( e.stopPropagation ) {
            e.stopPropagation();
        }
        e.cancelBubble = true;
    },
    stopImmediatePropagation: function() {
        this.isImmediatePropagationStopped = returnTrue;
        this.stopPropagation();
    },
    isDefaultPrevented: returnFalse,
    isPropagationStopped: returnFalse,
    isImmediatePropagationStopped: returnFalse
};
```

Figure 7.1: A fragment of jQuery (any version after 1.5.1).

Thus the two uses of `this` could refer to very different objects that just happen to both have a property called `originalEvent`. In practice, though, such a scenario is unlikely. Due to the function call semantics of JavaScript, the object on which a function is stored is often the `this` argument or a prototype thereof. Hence our analysis will determine that the two `originalEvent` tokens are related. By similar reasoning, the `isDefaultPrevented` tokens on lines 4 and 26 are also considered related.

Consider now `stopPropagation` on lines 12, 16, 17, and 24. The tokens on lines 16 and 17 are considered related, as are those on lines 12 and 24. But it is not evident whether these two groups are also related to each other. Two of them access a property on a jQuery event object or its prototype, while the other two access a property on `e`, which is an alias for `this.originalEvent`. There is no reliable indicator as to whether these refer to the same kind of object. The call on line 17 could be a recursive call to the function defined on line 12 or it could be a call to a lower-level event object whose method happens to have the same name. Our analysis considers the two groups of tokens unrelated from each other, and to rename the four tokens the programmer must thus answer two questions, one for each group.

In jQuery, the `originalEvent` property happens to refer to a native event object, created by the browser. As such, the `stopPropagation` tokens on lines 16 and 17 refer to a property on a native event object, while the tokens on lines 12 and 24 refer to a property on jQuery’s own event objects, which function as wrappers around the native event objects. Indeed, it is possible to rename the `stopPropagation` property on jQuery’s event objects by updating the tokens on lines 12 and 24 and one other token elsewhere.
in jQuery (outside the code shown in Figure 7.1), while leaving the tokens on lines 16 and 17 unaltered. The result is a refactored version of the jQuery library that remains compatible with most jQuery applications. Only applications that explicitly invoke stop-Propagation on a jQuery event object are incompatible with the new version. In this way, the programmer remains responsible for ensuring backward compatibility in case a renaming interferes with the public API of the library, but our technique makes it easier for the programmer to perform the desired refactorings of the library code.

Moreover, consider what would happen if we were to update every stopPropagation token in the previous example, that is, including those on lines 16 and 17. The call on line 17 would then no longer work as intended, because the browser would still use the name stopPropagation when creating native event objects, regardless of any source code transformation we have performed. As discussed previously, our technique does not perform safety checking, so we trust that the programmer does not attempt to perform such refactorings. This allows our technique to work without modeling the browser API.

A final concern when renaming a property is that all references to the property must be updated consistently. For instance, suppose we renamed only the stopPropagation token on line 12. Clearly, the call on line 24 would then fail. In this particular case, our technique groups these tokens together, and it is thus impossible for the programmer to perform the erroneous renaming using our refactoring tool. But this is not always the case; in general, the consistency of a renaming depends on the programmer answering correctly to the questions posed by the refactoring tool, which is no different than ordinary search-and-replace.

While our tool does not actively prevent inconsistent renamings, it should be able to carry out any consistent renaming that can be done by renaming of identifier tokens, assuming the programmer answers its questions correctly. To demonstrate the importance of this criterion, suppose our technique incorrectly determined that all four stop-Propagation tokens in Figure 7.1 were related. In this case, the previously mentioned refactoring of jQuery would not have been possible, because it involves renaming only a subset of these tokens. Indeed, if the programmer decided to rename the token on line 12, he would not get the opportunity to tell the tool not to rename those on lines 16 and 17, leading to a different refactoring that was intended. When working with a large codebase, the programmer might not immediately notice that his renaming did not go as intended, thereby introducing a bug in his code. Such behavior is therefore highly undesirable, and while our technique is not impervious to this type of failure, our evaluation shows that it is unlikely to occur in practice.

This example demonstrates that we need a static analysis that is able to approximate how object property tokens are related in JavaScript programs. This analysis must be sound in practice to avoid undesired refactorings, it must be sufficiently precise to enable a significant reduction of programmer effort compared to traditional search-and-replace, and it must be scalable and fast to be usable during programming. In addition, we want the analysis to work robustly on incomplete programs, such as libraries or applications under development. As we discuss in Section 7.5 no existing static analysis analysis satisfies these requirements, which motivates the design of our new analysis described in the following section.
7.3 Finding Related Identifier Tokens

Our proposed refactoring algorithm depends on a reliable method for finding related tokens in the program code. We define an ad-hoc type inference system, based on how objects appear to be used. Unlike most type systems, ours does not perform type checking, that is, there is no such thing as a static type error; all programs can be typed. A program typing is an equivalence relation between expressions, denoting which expressions have the same type. Given two occurrences of expressions, \( e_1.f \) and \( e_2.f \), we can subsequently say that the two \( f \) tokens are related if \( e_1 \) and \( e_2 \) have the same type. The type inference is considered sound if it is never the case that two tokens are found to be related while the programmer only intends to rename one and not the other.

The type inference is performed in two phases, denoted basic type inference and receiver type inference. Each phase traverses the abstract syntax tree once, generating a set of constraints. At the end of each phase, these constraints are saturated under a collection of closure rules using an augmented union-find data structure.

Our type analysis is reminiscent of alias analysis, in particular, the Steensgaard style \[82\], however, with important differences. Alias analysis determines which expressions in a program are aliased, that is, refer to the same object. The most common kind is may-alias analysis, which finds pairs of expressions that may, in some execution, evaluate to the same object. Given two expressions \( e_1.f \) and \( e_2.f \), one might consider the two \( f \) tokens to be related if \( e_1 \) and \( e_2 \) are may-aliased. However, may-alias analysis is conservative in the direction opposite of what we want: it may report some expression pairs as may-aliased even if they can, in fact, never be aliased. When this happens, unrelated tokens may be classified as related, which, as previously mentioned, is highly undesirable. Instead, one may consider must-alias analysis, which finds pairs of expressions that are aliases in every execution. This will result in a sound construction of related tokens, however, the resulting precision will be poor, even for a perfectly precise must-alias analysis. As an example, consider a composite expression \( x.f + y.f \) where \( x \) and \( y \) are aliased in some program executions but not in others. In this case, using a must-alias analysis will result in the two \( f \) tokens to be considered non-related. However, we would like to treat them as related, because there exists an execution where they refer to the same object property. This means that we wish to design a must-sometimes-alias analysis that finds pairs of expressions that must be aliases in some execution. We are not aware of existing analyses of that kind.

Moreover, our analysis must account for object prototypes, which JavaScript uses for mimicking inheritance, and certain common programming patterns, such as, prototype methods and object extension functions. For reasons explained in the following sections, these patterns indicate relatedness without involving aliasing, so we refer to our analysis as a type inference rather than an alias analysis.

7.3.1 Constraint Syntax

The constraints generated are of form \( t_1 \equiv t_2 \) where \( t_1, t_2 \) are terms. A term may either be an expression \( e \), or a compound \( e \circ f \) representing the \( f \) property on expression \( e \). The \( \equiv \) relation is called the same-type relation. We say that two terms, \( t_1 \) and \( t_2 \), have the same type when \( t_1 \equiv t_2 \). As an example, \( x \circ f \equiv y \) means that the variable \( x \) points to an object with an \( f \) property that has the same type as the variable \( y \).
The schema below summarizes the syntax and our naming conventions:

- \( o, v, e \in \text{expressions} \)
- \( t \in \text{terms} \)
- \( g \in \text{function bodies} \)
- \( t ::= e \mid e \circ f \)
- \( f \in \text{identifiers} \)
- \( \equiv \subseteq \text{terms} \times \text{terms} \)

We reserve the meta-variable \( o \) for object literals, \( v \) for program variables, and \( e \) for any type of expression. The notion of an expression is used quite liberally, but the meaning should be clear from the context: We use it primarily when referring to an occurrence of a JavaScript expression in the abstract syntax tree; thus, identical expressions at different positions in the program code are considered distinct. In a slight abuse of terminology, we also refer to variable declarations, such as a \texttt{var} statement or the token of a parameter name, as expressions. Finally, we include the following pseudo-expressions:

- \texttt{glob}: expression representing the global object
- \texttt{ret}(g): expression representing return value of function \( g \)
- \texttt{this}(g): expression representing \texttt{this} inside function \( g \)

Unlike expressions, the notion of an identifier does not denote a specific occurrence in the abstract syntax tree; two identifiers are indistinguishable if they consist of the same string of characters. We will use the notion of tokens when different occurrences of the same identifier should be considered distinct. The artificial identifier \texttt{[array]} refers to array entries.

### 7.3.2 Saturation

At the end of each phase, we saturate the \( \equiv \) relation until it satisfies a collection of closure rules. In particular, we ensure that \( \equiv \) is an equivalence relation:

\[
\begin{align*}
\frac{}{t \equiv t} \quad \text{(refl)} & \\
\frac{t_1 \equiv t_2}{t_2 \equiv t_1} \quad \text{(sym)} & \\
\frac{t_1 \equiv t_3}{t_2 \equiv t_3} \quad \text{(trans)} & 
\end{align*}
\]

and that \( \equiv \) moreover satisfies the following rule:

\[
\frac{e_1 \equiv e_2}{e_1 \circ f \equiv e_2 \circ f} \quad \text{(prty)}
\]

Informally, the prty rule states that same-typed expressions have same-typed properties. Examples in the next section demonstrate the consequence of the saturation rules. We present an efficient algorithm for performing the saturation in Section 7.3.5.

### 7.3.3 Basic Type Inference

In the first phase we traverse the abstract syntax tree and for each expression generate constraints according to Figure 7.2. We use the notation \( e_1 \equiv e_2 \equiv e_3 \) as shorthand for the two constraints \( e_1 \equiv e_2 \) and \( e_2 \equiv e_3 \). We also introduce the following two auxiliary definitions:

- \texttt{decl}(v): the declaration of \( v \), or \texttt{glob} \( \circ v \) if \( v \) is global
### 7.3. Finding Related Identifier Tokens

<table>
<thead>
<tr>
<th>statement or expression $e$</th>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable $v$</td>
<td>$e \equiv \text{decl}(v)$</td>
</tr>
<tr>
<td>property $e_1.f$</td>
<td>$e \equiv e_1 \circ f$</td>
</tr>
<tr>
<td>dynamic property $e_1[e_2]$</td>
<td>see text</td>
</tr>
<tr>
<td>*assignment $e_1 = e_2$</td>
<td>$e \equiv e_1 \equiv e_2$</td>
</tr>
<tr>
<td>*conditional $e_1 ? e_2 : e_3$</td>
<td>$e \equiv e_2 \equiv e_3$</td>
</tr>
<tr>
<td>*logical or $e_1</td>
<td></td>
</tr>
<tr>
<td>logical and $e_1 &amp;&amp; e_2$</td>
<td>$e \equiv e_2$</td>
</tr>
<tr>
<td>this</td>
<td>$e \equiv \text{this}(\text{fun}(e))$</td>
</tr>
<tr>
<td>return</td>
<td>$e_1 \equiv \text{ret}(\text{fun}(e))$</td>
</tr>
<tr>
<td>call</td>
<td>see text</td>
</tr>
<tr>
<td>new-call</td>
<td>see text</td>
</tr>
<tr>
<td>function function $f(\bar{v})$ { ... }</td>
<td>see text</td>
</tr>
<tr>
<td>array literal $[e_1, e_2, \ldots]$</td>
<td>$e \circ \text{[array]} \equiv e_1$</td>
</tr>
<tr>
<td>object literal { \ldots }</td>
<td>see below</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>member of object literal $o$</th>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializer $f : e_1$</td>
<td>$o \circ f \equiv e_1$</td>
</tr>
<tr>
<td>getter get $f { g \ldots }$</td>
<td>$o \circ f \equiv \text{ret}(g), \quad o \equiv \text{this}(g)$</td>
</tr>
<tr>
<td>setter set $f(v) { g \ldots }$</td>
<td>$o \circ f \equiv v, \quad o \equiv \text{this}(g)$</td>
</tr>
</tbody>
</table>

Figure 7.2: Constraints for basic type inference. A star * indicates that an exception to the rule is described in the text.

fun($e$): the innermost function containing $e$

We will now discuss each rule in detail.

**Variable** For an expression $e$ that is a variable $v$, we add the constraint that $v$ should have the same type as its declaration: $e \equiv \text{decl}(v)$. By transitivity, all uses of the variable will thereby have the same type.

**Example** The three uses of $v$ below will have same type as the declaration of $v$ on line 30 due to the variable rule. After saturation (Section 7.3.2), they will then all have the same type due to transitivity, and thus the two $x$ tokens will ultimately be considered related:

```plaintext
30 function f(v) {
31   v.x = v.y;
32   return v.x;
33 }
```

**Property** When a property expression $e$ of form $e_1.f$ is encountered, we add the constraint $e \equiv e_1 \circ f$. Due to the prty rule, if $f$ is accessed on a similarly typed expression, the two accesses will then be given the same type.

**Example** As per the previous example, the three uses of $v$ below are same-typed. We use subscripts to name tokens in the example code:

```plaintext
34 function f(v) {
35   v_{1}.x, y_{3} = v.y;
36   return v_{2}.x, y_{4};
37 }
```
7. Semi-Automatic Rename Refactoring for JavaScript

Because \( v_1 \equiv v_2 \), the prty rule yields the typing \( v_1 \odot x \equiv v_2 \odot x \). Since \( v_1 \odot x \equiv v_1 \odot x \) and \( v_2 \odot x \equiv v_2 \odot x \) were generated while traversing the abstract syntax tree, we get by transitivity that the two \( v \odot x \) expressions are same-typed. Thus, \( y_3 \) and \( y_4 \) will ultimately be considered related.

**Dynamic property** An expression of form \( e_1 [e_2] \) performs an array access or a reflective property access on \( e_1 \) (technically, they are the same in JavaScript). The name of the property being accessed depends on the value of \( e_2 \) at runtime. If \( e_2 \) is a string constant "f" we treat the expression as \( e_1.f \); in all other cases, we ignore the expression.

**Assignment** For an expression \( e \) of form \( e_1 = e_2 \), we add the constraints \( e \equiv e_1 \) and \( e \equiv e_2 \). There is an exception to this rule, however. Consider this chain assignment:

```javascript
38 x = y = null;
```

Such a statement is often employed as a compact way to clear the value of several variables, but is generally not a good indicator of the variables \( x \) and \( y \) having the same type. Indeed, no object could be assigned to both \( x \) and \( y \) by executing the statement. We classify certain expressions as *primitive* when they definitely cannot evaluate to an object. Null expressions are primitive, and an assignment expression is primitive if its right-hand side is primitive. If the right-hand side of an assignment is primitive, then we disregard the above rule for assignments and generate no constraints. No constraints are generated for compound assignment operators, such as, \( += \).

**Conditional** An expression \( e \) of form \( e_1 ? e_2 : e_3 \) evaluates \( e_1 \) and then evaluates and returns the value of either \( e_2 \) or \( e_3 \), depending on whether \( e_1 \) was true or false. We therefore add the constraints \( e \equiv e_2 \) and \( e \equiv e_3 \).

There is an exception to the above rule, however, since programmers occasionally use the \(?:\) operator in situations where an if-statement would have been appropriate. The following two statements are semantically equivalent:

```javascript
39 if (b) x = y else z = w;
40 b ? (x = y) : (z = w);
```

In the latter case, the result of the \(?:\) expression is immediately discarded. We say that such expressions occur in *void context*. When a \(?:\) expression occurs in void context, we disregard the rule above and generate no constraints. Otherwise, \( x \) and \( y \) would have been considered same-typed with \( z \) and \( w \) after saturation due to transitivity.

**Logical or** An expression \( e \) of form \( e_1 | e_2 \) will at runtime evaluate \( e_1 \), and then if \( e_1 \) is false, it will evaluate \( e_2 \) and return its result. If \( e_1 \) is true, the result of \( e_1 \) is returned. Although the operator is called *logical or*, its result need not be a boolean. Objects are considered to be true when coerced to a boolean value. Hence, an object from either \( e_1 \) or \( e_2 \) may be returned; we therefore add the constraints \( e \equiv e_1 \) and \( e \equiv e_2 \). As for the \( ?:\) operator, we disregard this rule when \( e \) occurs in void context.

**Logical and** An expression \( e \) of form \( e_1 && e_2 \) will at runtime evaluate \( e_1 \). If \( e_1 \) is true, it will then evaluate \( e_2 \) and return the result of \( e_2 \), and otherwise it will return the value of \( e_1 \). Since objects cannot be false when coerced to a boolean, only objects from \( e_2 \) may be returned. Thus, we add the constraint \( e \equiv e_2 \). The void context exception could be applied to this rule as well, but in this case it makes no difference, since \( e \equiv e_1 \) is not generated either way.
This For an expression $e$ of form `this` we add the constraint that $e$ should have the same type as the `this` argument in the enclosing function: $e \equiv this(fun(e))$. Thus, all uses of `this` in a given function will be given the same type.

Return For a return statement $e$ of form `return $e_1$, we add the constraint $e_1 \equiv ret(fun(e))$. This ensures that all returned expressions will have the same type.

Example In the function below, $a$ and $b$ will be same-typed because they are both returned within the same function. Thus the two `x` tokens will ultimately be considered related:

```javascript
function minX(a,b) {
    if (a.x < b.x) return a;
    else return b;
}
```

Call and new Most function calls are ignored by our algorithm. Precise inference of function types is complicated for a language such as JavaScript. If a call graph were available, we could connect arguments to parameters and return values to results, but if done in a context-insensitive manner, the transitivity of the $\equiv$ relation would in practice declare too many expressions as same-typed, which, as previously discussed, is highly undesirable. If done context-sensitively, scalability would be jeopardized. In our setting, the conservative action is to exclude constraints rather than include them, and as such, ignoring function calls can be tolerated.

One particular type of function call is easily handled, however. JavaScript programs often use `one-shot closures` to obtain encapsulation:

```javascript
(function(self) {
    var x; // 'x' not visible in outer scope
    /* ... */
})(this);
```

A function call $e$ of form $e_0(e_1,e_2,\ldots)$ in which $e_0$ is an anonymous function expression is easily handled for two reasons: (a) the called function is known, and (b) no other call can invoke that function. For this type of call, we add the constraint $e_i \equiv v_i$ for each corresponding argument $e_i$ and parameter $v_i$. Likewise, we add the constraint $ret(e_0) \equiv e$. If the call was made with the `new` keyword, we further add the constraint $this(e_0) \equiv e$, and otherwise $this(e_0) \equiv glob$ since the global object is passed as `this` argument in that case.

Example One-shot closures are often used together with for-loops as in the following example:

```javascript
for (var i=0; i<10; i++) {
    var panel = panels[i];
    var handler = (function(panel) {
        return function() {
            panel.activated(true);
        }
    })(panel);
    panel.button.addEvent("click", handler);
    panel.activated(false);
}
```

If the one-shot closure was not used, all ten event handlers would refer to the same `panel` variable, so the $i$th event handler would invoke `activated` on the last `panel`, rather than the $i$th panel. By handling one-shot closures, our analysis finds that the two uses of `panel`
on lines 53 and 57 have the same type, despite referring to different variables. Thus, the two activated tokens are ultimately considered related.

**Function** Functions are first-class objects in JavaScript, and thus may themselves have properties. One property of particular interest is the `prototype` property of a function object. When a function is invoked using the `new` keyword, a newly created object is passed as the `this` argument. This object has its internal prototype link initialized to the `prototype` property of the function object, effectively inheriting the properties of this object. For any function expression $e$ with body $g$, we therefore add the constraint $e \circ \text{prototype} \equiv \text{this}(g)$. For named functions, we similarly add the constraint $v \circ \text{prototype} \equiv \text{this}(g)$ where $v$ is the function name declaration.

Using the `prototype` property for a purpose other than inheritance is highly unusual, even for functions that are never invoked using `new`. Thus, the constraint will typically have no impact for functions that are not intended to be invoked using `new`.

**Example** In the code below, a string builder function is defined, and two functions are placed on its prototype object. Due to the above rule, `this` on line 60 will have the same type as `StringBuilder.prototype` on line 62, and the two `clear` tokens will thus be considered related:

```javascript
59 function StringBuilder() {
60     this.clear();
61 }
62 StringBuilder.prototype.clear = function() {
63     this.array = [];
64 }
65 StringBuilder.prototype.append = function(x) {
66     this.array.push(x);
67 }
68 StringBuilder.prototype.toString = function() {
69     return this.array.join("\n");
70 }
```

During the second phase of the algorithm, which we describe in Section 7.3.4, the three uses of `this` on lines 63, 66 and 69 will also get the same type, and the three array tokens will thus also become related.

**Array literal** An array literal $e$ of form $[e_1, e_2, \ldots]$ creates a new array object, initialized with the value $e_i$ in its $i$th entry. We assume such array objects are intended to be homogeneous, and add the constraint $e \circ [\text{array}] \equiv e_i$ for each $i$. A homogeneous array is an array for which all elements have the same type. Not all arrays are intended to be homogeneous, but we found that those created using array literals typically are.

Note that the artificial `[array]` property is not referenced by any of the other rules. In particular, there is no rule that handles array access expressions, since such expressions are hard to distinguish from reflective property accesses, and the array being accessed might not be homogeneous.

**Example** Array literals are often used to write out constant tables in JavaScript source code, as in the below snippet taken from Mozilla’s JavaScript PDF reader:

```javascript
71 var QeTable = [
72     {qe: 0x5601, nmps: 1, nlps: 1, switchFlag: 1}
73     ,{qe: 0x3401, nmps: 2, nlps: 6, switchFlag: 0}
74     ,{qe: 0x1801, nmps: 3, nlps: 9, switchFlag: 0}
75     /* ... */
76 ];
```
Since each member of the array is assigned the same type, each \texttt{qe} token will be considered related, and likewise for the three other property names.

**Object literal** An object literal \texttt{o} is an expression of form \{ ... \} containing zero or more object literal members. Such an expression creates a new object, with the object literal members denoting the initial values of its properties. There are three types of members: initializers, getters, and setters.

An initializer is of form \( f : e_1 \) and assigns the value of \( e_1 \) to the \( f \) property of the new object. For each such initializer, we add the constraint \( o \circ f \equiv e_1 \).

A getter is of form \texttt{get\ f\()\{ ... \}. This assigns a getter function for the \( f \) property on the new object. Whenever the \( f \) property is read from the object, the getter is invoked, and the value it returns is seen as the value of the \( f \) property. For each getter, we add the constraint \( o \circ f \equiv \text{ret}(g) \), where \( g \) denotes the getter function’s body. We also add \( o \equiv \text{this}(g) \), since the object is passed as \texttt{this} argument when the getter is invoked.

A setter is of form \texttt{set\ f\( (v)\{ ... \). This assigns a setter function for the \( f \) property. Whenever the \( f \) property is assigned to on the object, the setter is invoked with the new value passed as argument. We therefore add the constraints \( o \circ f \equiv v \) and \( o \equiv \text{this}(g) \), where \( g \) denotes the setter function’s body.

Getter and setters were introduced in ECMAScript 5 \[22\] and are now used in many applications; other recently introduced language features are ignored by our analysis.

### 7.3.4 Receiver Type Inference

In the second phase, we classify certain functions as \textit{methods} and add additional type constraints accordingly. However, reliably classifying functions as methods requires knowledge of \textit{namespace objects}, as we shall discuss shortly. We need the same-type relation inferred during the previous phase to detect such namespace objects, hence the division into two phases. We will now motivate the informal concepts of methods as namespaces, and then discuss the type inference algorithm for this phase.

JavaScript has no first-class concept of methods or constructors, only functions and invocation conventions that let programmers mimic these features. Function calls of form \( e. f(\ldots) \) will receive the value of \( e \) as the \texttt{this} argument, while calls of form \texttt{\new\ e(\ldots)} and \texttt{\new\ e.f(\ldots)} receive a newly created object as the \texttt{this} argument. Although programmers may mix and match these invocation types for any one function, most functions are in practice designed to be called exclusively one way or the other. Functions designed to be called with \texttt{\new} are referred to as constructors. Likewise, a non-constructor function stored in a property on some object is sometimes referred to as a \textit{method} of that object.

JavaScript also has no built-in support for namespaces, packages, or modules. This has led to a tradition of using objects to mimic namespaces. When a function is stored in some property of a namespace object, it may be wrong to consider it a method on the namespace, as it could just as well be a constructor.

Figure 7.3 demonstrates examples of methods and constructors stored on objects. To motivate the need to distinguish these cases, suppose the programmer has decided to rename the \texttt{active} token on line 78. For this type of code, it is generally safe to assume that the \texttt{active} token on line 81 is related to the one on line 78.

However, if instead the programmer had decided to rename the \texttt{x} token on line 84, it is not generally safe to assume that the \texttt{x} token on line 90 should be renamed as well. The
key difference is that Rect and Vector2 are constructors, and thus their this arguments are not related to the object on which the function is stored.

Unfortunately, the two cases are structurally quite similar. Given a statement of form \texttt{\textit{e}.\textit{f}=\textit{function}(\ldots)\{\ldots\}}, we cannot immediately tell if the function is a method or a constructor.

In this phase, we exploit two indicators of a function being a constructor. The most direct indicator is that it is invoked using the \texttt{new} keyword. The second indicator is that the source code mentions its \texttt{prototype} property. As previously mentioned, this property has a special meaning when used on functions, and is typically only used for constructor functions.

Generally detecting how a function is called is a hard problem involving call graph construction, but our situation is much simpler than that. We are primarily interested in functions that are accessed through namespace objects, which we can easily recognize statically using the following heuristic.

### Namespace detection
After the constraints from the prior phase have been generated, we saturate the \texttt{=} relation under the closure rules in Section 7.3.2. Then, for any expression of form \texttt{\texttt{new} \textit{e}.\textit{f}(\ldots)} or \texttt{\textit{e}.\textit{f}.prototype} we mark the type of \textit{e} as a \texttt{namespace} type. Any expression with same type as \textit{e} will be considered to be a namespace. Namespaces are not considered to have \texttt{any} methods.

Using the above heuristic, the Rect function in Figure 7.3 will not be considered a method on the util object, because util has been marked as a namespace by the \texttt{new} call on line 94.

There is an inherent whole-program assumption in this heuristic. If the programmer has created a namespace with two functions intended to be used as constructors, but is not yet using any of them, the heuristic may fail to detect the namespace object. For example, if the \texttt{new} call on line 94 were not yet written, the two x tokens in Figure 7.3(b) would be treated as being related. The practical implications of this are discussed in the evaluation section.

### Method definition
Once namespaces have been identified, we look for \textit{method definitions}. For an expression of form \texttt{\textit{e1}.\textit{f}=\textit{e2}} where \textit{e1} was not marked as a namespace and \textit{e2} is a function expression, we say \textit{e2} is a \textit{method definition} with \textit{e1} as its \textit{host} expression.
7.3. Finding Related Identifier Tokens

Likewise, for an initializer \( f : e_2 \) inside an object literal \( o \), we say \( e_2 \) is a method definition with \( o \) as its host expression. For any method definition with body \( g \) and host expression \( e \), we add the constraint \( e \equiv \text{this}(g) \).

**Example (prototype method)** In the code below, the `baz` function is considered a method on `Foo.prototype`, hence its `this` argument will get the same type as `Foo.prototype`. Since `this` inside the `Foo` function also has the same type as `Foo.prototype` by the function rule from the prior phase, the two uses of `this` have the same type. Hence, the two uses of `x` will ultimately be considered related.

```javascript
95 function Foo(x1) {
96   this.x = x1;
97 }
98 Foo.prototype.baz = function() {
99   alert(this.x);
100 };
101 new Foo(5).baz(); // alerts "5"
```

**Example (extend function)** The code below uses the `extend` function commonly found in third-party libraries, which copies all properties from one object to another. The host object of the `baz` method is thus a temporary object that exists only briefly until its properties have been copied onto the `Foo` object. Even though this temporary object is never actually passed as `this` to the `baz` method on line 105, the method definition constraint ensures that the object literal has the same type as `this`, which has the desired effect: the two uses of `x` will be considered related.

```javascript
102 var Foo = {}; 103 Object.extend(Foo, { 104   x: 5, 105   baz: function() { 106     alert(this.x); 107   } 108 });
109 Foo.baz(); // alerts "5"
```

This example, as well as the following one, also demonstrates why our analysis is technically not an alias analysis: the `this` expression on line 106 is not an alias of the object literal, so if we used a precise alias analysis instead of our type inference, the two uses of `x` would not be considered related.

**Example (class system)** The code below uses a popular class system provided by the prototype.js library to simulate the creation of a class. As with the `extend` function, the host object for the two methods is in fact not passed as `this` to either method, but again, receiver type inference has the desired effect: the two uses of `x` become related.

```javascript
110 var Foo = Class.create({ 111   initialize : function(x1) { 112     this.x = x1;
113   }, 114   baz : function() { 115     alert(this.x);
116   } 117 });
118 new Foo(5).baz(); // alerts "5"
```

[http://prototypejs.org/](http://prototypejs.org/)
7.3.5 Saturation Algorithm

At the end of each phase, the $\equiv$ relation is saturated until it satisfies the closure rules in Section 7.3.2. Since the result is an equivalence relation, it can be represented efficiently using a union-find data structure [16]. In the following, we assume the reader is familiar with union-find and associated terminology.

The only rule not immediately satisfied by virtue of the traditional union-find algorithm is the $\text{prty}$ rule. We augment the union-find data structure such that each node has a $\text{prty}$ field, in addition to the standard parent pointer and rank fields. The $\text{prty}$ field holds a map from strings (property names) to nodes of the union-find data structure. Initially, one union-find node is created for each program expression, including the pseudo-expressions defined in Section 7.3.1. We informally refer to these nodes as types, since two expressions have the same type exactly if their nodes have a common representative. The type of a term $e \circ f$ is the node pointed to by the $f$ entry in the $\text{prty}$ map of the representative of $e$. If no such node exists, then $e \circ f$ is not same-typed with any other term.

When two nodes $n_1, n_2$ are unified, such that $n_1$ becomes the new root, their $\text{prty}$ maps are merged by the following procedure. For property names present only in $n_2$’s $\text{prty}$ map, the corresponding entry is copied over to $n_1$’s $\text{prty}$ map. For property names present in both maps, the nodes they refer to are recorded in a worklist of node-pairs that should be unified. As a simple and effective optimization to this procedure, we initially swap $n_1$ and $n_2$’s $\text{prty}$ pointers if $n_2$’s $\text{prty}$ map is bigger than $n_1$’s, so fewer entries need to be copied. At the end of each phase, node pairs are removed from the worklist and unified until the worklist is empty.

We also store an $\text{isNamespace}$ boolean on each node, for marking nodes as namespaces during the second phase. The namespace detection can be done while traversing the abstract syntax tree in the first phase.

A pseudo-code implementation of the augmented union-find data structure is given in Figure 7.4. The code does not include the generation of constraints during traversal of the abstract syntax tree.

When a constraint of form $e_1 \equiv e_2$ is discovered, we invoke the $\text{unify}$ method with the corresponding nodes. For a constraint of form $e_1 \circ f \equiv e_2$, we invoke $\text{unifyPrty}$ instead. The constraints need not be stored explicitly; invoking the corresponding method on the $\text{Unifier}$ instance is sufficient. At the end of each phase, we invoke the $\text{complete}$ method to ensure that the $\text{prty}$ closure rule is satisfied.

7.4 Evaluation

We implemented a renaming plugin\footnote{http://www.brics.dk/jsrefactor/plugin.html} for Eclipse, based on the algorithm described in the previous section. We use the same underlying implementation for this evaluation.

Ideally, the primary metric of usefulness of a refactoring tool is its impact on programmer productivity. This is unfortunately hard to define and difficult to measure directly, so we base our evaluation on the following more tangible metrics, which we consider good indicators of usefulness:
7.4. Evaluation

class UnifyNode:
    field parent = this
    field rank = 0
    field prty = <empty map>
    field isNamespace = false

    def rep():
        if parent != this:
            parent = parent.rep()
        return parent

class Unifier:
    field queue = <empty queue>

    def unify(x, y):
        x = x.rep()
        y = y.rep()
        if x == y:
            return
        if x.rank < y.rank:
            swap x, y
        else if x.rank == y.rank:
            x.rank += 1
            y.parent = x
            x.isNamespace |= y.isNamespace
        if x.prty.size < y.prty.size:
            swap x.prty, y.prty
        for k, v in y.prty:
            if k in x.prty:
                unifyLater(x.prty[k], v)
            else:
                x.prty[k] = v
        y.prty = null

    def unifyPrty(x, k, y):
        x = x.rep()
        if k in x.prty:
            unify(x.prty[k], y)
        else:
            x.prty[k] = y

    def unifyLater(x, y):
        queue.add(x, y)

    def complete():
        while queue is not empty:
            (x, y) = queue.pop()
            unify(x, y)

Figure 7.4: Python-like pseudo-code implementation of the augmented union-find data structure.
Manual Effort: How many questions must the programmer answer to complete a renaming? Since the programmer need only consider a single token per question, a question issued by our tool is no more difficult than the corresponding search-and-replace question.

Soundness: If given correct answers by the programmer, how likely is it that a renaming is ultimately inconsistent? In other words, is the analysis sound in practice?

Delay: How long must the programmer sit idle while waiting for the tool to finish a computation?

Whole-Program: Does the tool apply to library code, without having application code available? Does it apply to incomplete application code, such as, code under development or applications without libraries?

We collected 24 JavaScript applications for use as benchmarks. Third-party libraries are used in 19 of the applications, constituting a total of 9 distinct libraries (some libraries are used by more than one application) that we also include as benchmarks. Of these 24 applications, 10 were taken from the 10k Event Apart Challenge,10 were taken from Chrome Experiments,6 and 4 were found at GitHub.7 When selecting benchmarks, we aimed for diversity in complexity, functionality, and use of libraries. The benchmark collection is available online.8

To evaluate our technique along the above metrics, we perform a series of experiments on these applications. The benchmarks and experimental results are shown in Table 7.1. In the following, we describe which experiments were used to evaluate the various metrics, and the meaning of the columns in Table 7.1.

7.4.1 Manual Effort

For both our technique and search-and-replace, the total number of questions asked during a refactoring is independent of whether the programmer answers yes or no to each question. It is also independent of the new name for the identifier. This means there is a simple way to compute the number of questions each tool will ask, given the name of the property to rename. For our approach, we can count the number of groups of related tokens with the given name, and for search-and-replace we can count the number of tokens. Tokens that refer to local variables are ignored in this statistic.

Renaming tools are typically invoked by selecting an identifier token in the editor and then choosing the rename action from a menu or using a shortcut. There is no reason to ask a question for this initially selected token, so we subtract one question per property name to represent this question that is answered for free. Thus, the tool may potentially ask zero questions to complete a refactoring. However, we shall disregard properties that are only mentioned once in the source code, so search-and-replace will by design always ask at least once. We also disregard the special property name prototype since it is easily recognized as a built-in property and is thus not renamed in practice.

7.4. Evaluation

To measure effort, we compare the number of questions issued per benchmark by search-and-replace versus our approach, simulating one rename refactoring on each distinct property name appearing in the benchmark. Application developers are unlikely to want to rename properties inside third-party libraries, and vice versa, so we separate the set of renaming tasks for application code and library code. However, we allow our refactoring tool to analyze the application and library code together as a whole, to avoid interference with the whole-program metric that we address in Section 7.4.4.

In Figure 7.5, each simulated renaming is plotted as a point positioned according to the number of questions asked by search-and-replace and by our tool, respectively. The diagonal highlights the place where the two tools are equally effective. The plot has been clamped to a limit of 100 questions, since the majority of points lie in this range. The most extreme outliers excluded from the plot are the property names \( c \) and \( f \) in pdfjs, which require 6,092 questions with search-and-replace and 14 questions with our tool. A total of 4,943 renamings were simulated in this experiment. Figure 7.6 shows how tokens are distributed in groups of various sizes during the simulated renamings.

To estimate the effect per benchmark, we sum the total number of questions for each property name in each benchmark. In Table 7.1, the effect column denotes the reduced programmer effort, based on this estimate. It is computed using this formula:

\[
effect = \frac{\text{#search-replace-questions} - \text{#our-questions}}{\text{#search-replace-questions}}
\]

Thus, if our tool asks the same number of questions as search-and-replace, we get an effect of 0%, while if we ask no questions at all, we get an effect of 100%.

The effect is in most cases between 40% and 70% but reaches 17% and 97% in extreme cases. The average and median effect are both approximately 57.1%. Two thirds of the benchmarks show an effect of more than 50%.
Each library that is used by multiple applications is analyzed together with each client application. In each case, we observe that the effect is exactly the same regardless of which application is used.

Although this indicates a significant reduction in the effort required to perform renamings, it may be that the number of questions remains so high that programmers deem the refactoring too costly and choose to omit an otherwise desirable renaming altogether. If we regard renaming tasks that require more than three questions as hard and renaming tasks that require at most three as easy, we can consider how many renaming tasks are easy when using search-and-replace versus our tool.

Figure 7.7 (left) shows how many properties can be renamed using at most three questions. All benchmarks benefit from the new technique, and in many cases we observe a substantial increase in the number of easy renamings. In Figure 7.7 (right) we focus
### 7.4. Evaluation

**Applications**

<table>
<thead>
<tr>
<th>Applications</th>
<th>lines</th>
<th>libraries used</th>
<th>whole effect</th>
<th>isolated effect</th>
<th>fragmented effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>?spade</td>
<td>959</td>
<td></td>
<td>29.3%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>airstrike</td>
<td>1,508</td>
<td>prototype</td>
<td>68.5%</td>
<td>-</td>
<td>-0.03 pp</td>
</tr>
<tr>
<td>bpmster</td>
<td>990</td>
<td>jquery</td>
<td>49.6%</td>
<td>-0.41 pp</td>
<td>-0.63 pp X</td>
</tr>
<tr>
<td>bytesjack</td>
<td>685</td>
<td>jquery, typekit</td>
<td>52.7%</td>
<td>-0.15 pp</td>
<td>-</td>
</tr>
<tr>
<td>film-buffr</td>
<td>203</td>
<td>jquery, typekit</td>
<td>29.3%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>inspiration</td>
<td>575</td>
<td>jquery, typekit</td>
<td>35.1%</td>
<td>-0.70 pp</td>
<td>-1.60 pp</td>
</tr>
<tr>
<td>phrase</td>
<td>176</td>
<td>jquery, typekit</td>
<td>60.4%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>space-mahjong</td>
<td>344</td>
<td>jquery, typekit</td>
<td>17.2%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sun-calc</td>
<td>589</td>
<td></td>
<td>67.1%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>top-of-the-world</td>
<td>158</td>
<td>jquery, typekit</td>
<td>49.1%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ball-pool</td>
<td>347</td>
<td>box2d, prototype</td>
<td>55.5%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fluid-simulation</td>
<td>484</td>
<td></td>
<td>97.0%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>google-gravity</td>
<td>6,621</td>
<td></td>
<td>71.1%</td>
<td>-</td>
<td>-0.05 pp</td>
</tr>
<tr>
<td>htmleditor</td>
<td>1,835</td>
<td>codemirror, esprima</td>
<td>43.9%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>minesweeper</td>
<td>562</td>
<td>jquery</td>
<td>90.8%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>procedural-flower</td>
<td>2,215</td>
<td>dat.gui</td>
<td>67.9%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sketchtoy</td>
<td>959</td>
<td>jquery, show_ads</td>
<td>61.0%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>texter</td>
<td>246</td>
<td>dat.gui</td>
<td>64.3%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>webgl-water</td>
<td>1,627</td>
<td></td>
<td>57.4%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ztype</td>
<td>3,581</td>
<td>show_ads</td>
<td>72.3%</td>
<td>-</td>
<td>-0.06 pp</td>
</tr>
<tr>
<td>impress.js demo</td>
<td>448</td>
<td>twitter_widgets</td>
<td>57.1%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jsgb</td>
<td>3,815</td>
<td>show_ads</td>
<td>44.9%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jslinux</td>
<td>10,054</td>
<td></td>
<td>80.4%</td>
<td>-</td>
<td>-0.27 pp</td>
</tr>
<tr>
<td>pdfjs</td>
<td>38,958</td>
<td></td>
<td>72.5%</td>
<td>-</td>
<td>-0.02 pp</td>
</tr>
</tbody>
</table>

**Libraries**

<table>
<thead>
<tr>
<th>Libraries</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jquery</td>
<td>6,098</td>
<td></td>
<td>48.4%</td>
<td>-</td>
<td>-0.03 pp X</td>
</tr>
<tr>
<td>typekit</td>
<td>1,359</td>
<td></td>
<td>51.1%</td>
<td>-</td>
<td>-0.04 pp</td>
</tr>
<tr>
<td>prototype</td>
<td>4,954</td>
<td></td>
<td>48.7%</td>
<td>-</td>
<td>-0.15 pp</td>
</tr>
<tr>
<td>box2d</td>
<td>6,081</td>
<td></td>
<td>72.1%</td>
<td>-</td>
<td>-0.03 pp</td>
</tr>
<tr>
<td>esprima</td>
<td>3,074</td>
<td></td>
<td>63.9%</td>
<td>- (X)</td>
<td>- (X)</td>
</tr>
<tr>
<td>codemirror</td>
<td>5,008</td>
<td></td>
<td>52.0%</td>
<td>-</td>
<td>-0.03 pp</td>
</tr>
<tr>
<td>dat.gui</td>
<td>2,149</td>
<td></td>
<td>58.8%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>show_ads</td>
<td>590</td>
<td></td>
<td>54.1%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>twitter_widgets</td>
<td>2,965</td>
<td></td>
<td>41.7%</td>
<td>-</td>
<td>-0.22 pp</td>
</tr>
</tbody>
</table>

Table 7.1: Experimental results. The effect column denotes reduced programmer effort (higher is better). The $\Delta$effect columns are in percentage points (pp), relative to the effect column (zero is better). An ‘X’ denotes a potential failure discussed in the text. The symbol ‘-’ indicates zero.
7. Semi-Automatic Rename Refactoring for JavaScript

on the property names for which search-and-replace requires more than three questions. This plot demonstrates that many of the renamings that are hard when using search-and-replace are easy when using our tool. For two thirds of the benchmarks, more than half of the hard renamings become easy.

In Figure 7.8 we vary the number of questions allowed, while focusing on two benchmarks, one with low and one with high effect. While the bound of three questions in Figure 7.7 was arbitrary, these figures indicate similar improvements for other bounds as well.

These figures indicate that even programmers who are unwilling to invest much time in renaming tasks may benefit from our tool. In summary, this part of the evaluation indicates a substantial reduction of programmer effort compared to traditional search-and-replace.

7.4.2 Soundness

As described in Section 7.3, our program analysis is theoretically unsound, but designed to work soundly on common programming patterns. We define a failure as a situation where unsoundness occurs in practice, that is, where our technique classifies two identifier tokens as related, even though they were not intended to refer to the same property. To estimate how likely this is to happen, we used a combination of dynamic analysis and manual inspection.

We exercise instrumented versions of each application, including the libraries they use, and record aliasing information at runtime. In these executions, we manually provide user input to each application, aiming to use all of its features at least once. High coverage is desirable, as it reduces the amount of manual inspection we must do afterward.

Given two expressions $e_1 \cdot f$ and $e_2 \cdot f$, if $e_1$ and $e_2$ evaluate to the same object at some point in the execution, then we have strong evidence to support that the two $f$ tokens should indeed be related. If our technique classifies these as related, we therefore consider the classification as correct.

In situations where our tool classifies two tokens as related, but the dynamic alias information does not provide evidence supporting this relationship, we resort to manual...
inspection. This may happen because of incompleteness in the concrete execution, or more commonly, because of meta-programming patterns in which properties are copied between objects, which our dynamic analysis cannot detect.

This experiment uncovers a single potential failure, which takes place in the \textit{esprima} library—a JavaScript parser written in JavaScript. This library represents abstract syntax trees with objects, and all such objects have a \texttt{type} property denoting the type of the node it represents. Different types of nodes occasionally have same-named properties; a property name, such as, \texttt{value} is quite common among these. However, an \textit{esprima} developer might hypothetically want to rename the \texttt{value} property for some AST nodes, but not all of them. Such a renaming could be done consistently, but our technique will not permit such a refactoring, because differently typed AST nodes are occasionally returned from within the same function, hence regarded as same-typed. This potential failure is marked with an X in the leftmost \textit{failure} column in Table 7.1. Except for this single case, the experiments confirm that our analysis is sound in practice when the complete program code is available.

### 7.4.3 Delay

For each application, we measure the time required to analyze the application code together with its library code. We analyze each benchmark eleven times, discard the timings from the initial warm-up run, and record the average time of the other ten runs.

The analysis takes less than one second in every case, with 780 milliseconds for \textit{pdfjs} being the slowest. Altogether, our implementation handles an average of around 50,000 lines per second on a 3.00 GHz PC running Linux.

When the user initiates a renaming, he must first enter a new name for the renamed property. The parsing and analysis do not depend on this new name and can therefore be performed in the background while the user is typing. As long as the user takes more than one second to type in the new name, there should therefore be no observable delay in practice when using our renaming technique.

### 7.4.4 Whole-Program

For the experiments described in the preceding sections, our static analyzer has an entire application available. However, library developers will typically not have a range of client applications ready for when they want to rename something.

To measure how well our tool works for library code, we repeat the experiment from Section 7.4.1 except that each library is now analyzed in isolation. As result, we observe no difference in the \textit{effect} for any library compared to the previous experiment, in which the application code was included. This is indicated in the \textit{isolated/\Delta effect} column of Table 7.1.

For completeness, we also did the converse experiment: we analyze the applications without their libraries, even though application developers will typically have a copy of their libraries available. For three applications, the effect diminishes slightly, but never by more than a single percentage point. In the most pronounced case, the effect drops from 35.1\% to 34.6\% in the \textit{nspiration} application.

These experimental results indicate that our approach is effective without a whole-program assumption.
We also want to support refactoring of code that is under development, which we call *incomplete* code. Such code might have radically different characteristics than finished code.

To estimate how well our tool works on incomplete code, once again we repeat the experiment from Section 7.4.1 except now with random pieces of code removed from each benchmark. We say the source code has been *fragmented*. Concretely, we replace random function bodies with empty bodies, which simulates incomplete code while avoiding introduction of syntax errors. For each benchmark, we produce ten fragmented versions randomly, each being roughly half the size of the original version. For each fragmented version, we then compute groups of related identifier tokens and compare these with the corresponding groups from the original version. We then record which token pairs were no longer considered related after fragmentation and compute the difference in effect accordingly (while only considering tokens that were not removed).

As a result, the effect diminishes for seven applications and six libraries, in each case only slightly. The largest change is for a fragmented version of *inspiration*, in which the effect drops from 69.8% to 66.7% (again, for identifier tokens that were not removed in the fragmentation). The *fragmented/∆effect* column in Table 7.1 shows the reduction in effect, averaged over the ten fragmented versions. The numbers demonstrate that our analysis is robust, even when major parts of the code are omitted.

This experiment also exposes a type of unsoundness caused by our namespacedetection mechanism discussed in Section 7.3.4. The benchmarks *bpmster* and *jquery* both contain constructor functions in namespace objects. After fragmentation, all the *new* calls that allowed us to classify these functions as constructors were occasionally deleted, causing the receiver type inference phase to treat the constructors as methods. This in turn causes some unrelated tokens to be classified as related. As discussed in Section 7.3.4, the namespace detection uses a whole-program assumption, which surfaces in these few cases where the relevant code is not available to the analysis. In principle, these failures could be averted by allowing the programmer to provide a single namespace annotation to each benchmark, marking a few global variables as namespaces.

### 7.4.5 Threats to Validity

The validity of these encouraging experimental results may be threatened by several factors. Most importantly, the simulated refactorings may not be representative of actual use cases. An alternative evaluation approach would be to conduct extensive user studies, but that would be major endeavour in itself, which is also the reason why most literature on refactoring algorithms settle for automated experiments.

The fragmented code we produce might not be representative of code under development. Revision histories from version control systems might provide a more faithful representation of incomplete code, but we did not have access to such data for most of our benchmarks.

Finally, our selection of benchmarks might not represent all mainstream JavaScript coding styles, although they have been selected from different sources and vary in complexity, functionality, and use of libraries. All our benchmarks are browser-based. However, our approach works without any model of the host environment API, so it seems reasonable that it should work well also for other platforms.
7.5 Related Work

In previous work [27], we showed that points-to information can be used to perform various refactorings on JavaScript programs, fully automatically and with safety checks to ensure that program behavior is preserved. That technique, however, has some practical limitations regarding scalability and whole-program assumptions of the points-to analysis, as discussed in the introduction. Our new more pragmatic approach requires more manual effort from the user, but applies to a wider range of codebases.

Fast alias analyses in the style of Steensgaard [82] use union-find data structures to efficiently find aliased expressions in a program. Our type inference system is inspired by this kind of analysis, but as discussed in Section 7.3, it is technically not an alias analysis, since we occasionally want non-aliased expressions to have the same type.

Several related static analysis techniques have been developed specifically for JavaScript. The type analysis by Jensen et al. [46] is based on flow-sensitive dataflow analysis, Var- douakis [87] uses a pushdown flow analysis to infer type information, and the flow analysis by Guha et al. [41] relies on reasoning about control flow. The notion of types we use here is different, and the application of our analysis is not type checking but refactoring. Sridharan et al. [80] have made advancements toward analyzing library code with points-to analysis, and Madsen et al. [59] have devised a practical technique for analyzing application code without including library source code or modeling external APIs. None of all these approaches scale to JavaScript programs of the size we consider here. Moreover, these analyses are conservative in the direction opposite of what we want: what they classify as spurious flow translates into unsoundness in our setting, and vice versa.

7.6 Conclusion

We have presented a technique for semi-automatic refactoring of property names in JavaScript programs, based on a static analysis for finding related identifier tokens. The analysis is easy to implement, and our experiments demonstrate that it is fast enough to be usable in IDEs. By simulating renaming tasks, our technique reduces the manual effort required to rename object properties by an average of 57% compared to the search-and-replace technique in existing JavaScript IDEs. This substantial improvement diminishes only slightly when used on incomplete code, and seemingly not at all when used on libraries without client code.

Although the analysis is theoretically unsound, the experiments show that it is sound in practice, except for a few rare cases, most of which could be eliminated entirely with simple source code annotations.

Our technique is based on a set of typing rules reminiscent of alias analysis. While we have focused on renaming so far, we would like to explore further applications of these typing rules in future work, such as, other refactorings, code completion, and documentation, to provide additional IDE support for JavaScript programmers.

Acknowledgments We thank Max Schäfer for his comments on a draft of this paper.
Checking Correctness of TypeScript Interfaces for JavaScript Libraries

This chapter is a copy of an OOPSLA’14 article [25] by Asger Feldthaus and Anders Møller.

Abstract

The TypeScript programming language adds optional types to JavaScript, with support for interaction with existing JavaScript libraries via interface declarations. Such declarations have been written for hundreds of libraries, but they can be difficult to write and often contain errors, which may affect the type checking and misguide code completion for the application code in IDEs.

We present a pragmatic approach to check correctness of TypeScript declaration files with respect to JavaScript library implementations. The key idea in our algorithm is that many declaration errors can be detected by an analysis of the library initialization state combined with a light-weight static analysis of the library function code.

Our experimental results demonstrate the effectiveness of the approach: it has found 142 errors in the declaration files of 10 libraries, with an analysis time of a few minutes per library and with a low number of false positives. Our analysis of how programmers use library interface declarations furthermore reveals some practical limitations of the TypeScript type system.

8.1 Introduction

The TypeScript [60] programming language is a strict superset of JavaScript, one of the main additions being optional types. The raison d’être of optional types is to harness the flexibility of a dynamically typed language, while providing some of the benefits otherwise reserved for statically typed languages: for instance, the compiler can quickly catch obvious coding blunders, and editors can provide code completion. Types also carry documentation value, giving programmers a streamlined language with which to document APIs. TypeScript does not ensure type safety, in other words, the type system is unsound by design. Even for programs that pass static type checking, it is possible that a variable
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

at runtime has a value that does not match the type annotation. This is seen as a trade-off necessary for keeping the type system unrestrictive.

TypeScript is designed to compile to JavaScript using a shallow translation process. Each TypeScript expression translates into one JavaScript expression, and a value in TypeScript is the same as a value in JavaScript. This makes it possible to mix TypeScript code with JavaScript code, in particular existing JavaScript libraries, without the use of a foreign function interface as known from other programming languages. However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript. Once such a description has been written for a JavaScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository\footnote{https://github.com/borisyankov/DefinitelyTyped} which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may reject perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes no attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it can easily remain undetected long enough for the buggy declaration file to get published to the community. Hence, application programmers who depend on the declaration file may occasionally find themselves debugging their own code, only to find that the bug was in the declaration file.

Figure 8.1 shows an example of a bug found in the declaration file for the d3.js library. The initialization code for the library creates the d3 and d3.scale objects, and exposes the threshold function on d3.scale. The library uses the d3 object as a namespace for all its exposed features. Note that the function expression spanning lines 1–13 is invoked immediately on line 13, exposing the d3_scale_threshold function (and other features not shown in the fragment).

The declaration file declares a module D3 to provide a namespace for type declarations, and declares that the global variable d3 must satisfy the type D3.Base, which refers to the interface on line 21. The interface D3.Base declares a property scale of type Scale.ScaleBase, which refers to D3.Scale.ScaleBase defined on line 16. It follows that the d3.scale object created during library initialization must satisfy the D3.Scale.ScaleBase interface. This interface declares a threshold method, which is intended to document the function created on line 0, but the declaration file contains a misspelling of the word “threshold”. This has consequences for users of the declaration file: First, editors that provide code completion (also called autocompletion) will suggest that an expression like d3.scale.th is completed to d3.scale.theshold(), which will pass the static type check, but fail at runtime. Second, if the programmer discovers the problem and fixes the typo in his client code, the static type checker will reject his otherwise correct code, claiming that ScaleBase has no method called threshold.
8.1. Introduction

Fragment of d3.js (library implementation)

```javascript
1 var d3 = function() {
2   var d3 = {...};
3   ...
4   d3.scale = {};
5   ...
6   d3.scale.threshold = function() {
7      return d3.scale.threshold([.5], [0, 1]);
8   };
9   function d3_scale_threshold(domain,range) {
10      ...
11   } return d3;
12 }();
```

Fragment of d3.d.ts (declaration file)

```typescript
14 declare module D3 { ...
15   module Scale { ...
16      interface ScaleBase { ...
17         threshold(): ThresholdScale;
18      }
19      interface ThresholdScale { ... }
20   }
21      interface Base { ...
22         scale : Scale.ScaleBase;
23   }
24 }
25 declare var d3 : D3.Base;
```

Figure 8.1: Fragments of the d3.js library and the associated declaration file found on the Definitely Typed repository. The ellipses (“...”) represent fragments taken out for brevity.

Some declaration files, including the one for d3.js, consist of over a thousand effective lines of code and are not trivial to write or proofread. Currently, the main line of defense against declaration bugs is to write tests using a code completing editor, deliberately using the completion suggestions. These tests invoke certain features of the library, and if the tests both type check and do not fail at runtime, the declaration file is likely to be correct for those features. An obvious downside to this approach is that the programmer effectively has to duplicate parts of the library’s test suite (which is not written in TypeScript, and hence cannot be checked by the compiler). Such tests are present for many libraries in the Definitely Typed repository, but they are incomplete and many bugs have escaped the scrutiny of these tests. For instance, there are 1,800 lines of code for testing the declaration file for d3.js, yet the aforementioned bug was not caught.

The API of a JavaScript library is not expressed declaratively as in other languages but defined operationally by its initialization code that sets up the objects and functions that constitute the library interface. This initialization is typically done during execution of the top-level code of the library, that is, immediately when the JavaScript file is loaded and not later when events are processed. Based on this observation, we expect that many mismatches between TypeScript declarations and JavaScript implementations can be detected by considering the effects of the library initialization code.

In this work we present a technique for checking the correctness of TypeScript declaration files against a JavaScript implementation. Our technique proceeds in three phases:
1. We first execute the library’s initialization code and extract a snapshot of its state.

2. Next we check that the global object and all reachable objects in the snapshot match the types from the declaration file using a structural type checking algorithm.

3. We then complement the preceding phase with a light-weight static analysis of each library function to check that it matches its signature in the declaration file.

The error in Figure 8.1 is one of those found by type checking the heap snapshot (phase 2). We shall later see examples of errors that are caught by the static analysis of the function code (phase 3).

With our implementation, tscheck, we observe that many large declaration files published to the community contain bugs, many of which cause other tools to misbehave. We also observe that a large portion of the bugs are easily fixed, while others are harder to address due to limitations in TypeScript’s type system.

In summary, our contributions are as follows:

- We point out the need for tool support to check correctness of TypeScript declaration files against JavaScript library implementations, and we present an algorithm that is designed to detect many kinds of errors in such declarations. Our algorithm checks that the runtime object structure after library initialization and the library function code are consistent with the given declaration file.

- As part of describing the algorithm, we formalize the relationship between values and types in TypeScript, which is otherwise only described informally in the language specification.

- Our experimental results show that the algorithm is capable of detecting 142 errors in a collection of 10 existing declaration files, with a low number of false positives. The snapshot type checking phase takes less than a second to execute for each library, and the static analysis checks each function in about one second.

- The development of our algorithm and experiments has exposed some practical limitations of TypeScript’s type system, which may be useful information for evolution of the TypeScript language.

Our algorithms naturally depend on what it means for a runtime value to have a certain type according to the TypeScript language specification. The snapshot type checking phase described in Section 8.4 is designed to be sound and complete with respect to this meaning of types, modulo two assumptions we explain in Section 8.3 about library initialization and two deviations that we introduce for reasons explained in Section 8.4.1. As mentioned, TypeScript’s static type checker is deliberately unsound, as exemplified by the following program:

```javascript
26 var x : string = "foo";
27 var y : any = x;
28 var z : number = y;
```

The assignment on line 28 would require a type cast in a Java-like type system (which would consequently fail at runtime), but the assignment is deliberately allowed in TypeScript. Our algorithm for statically analyzing the function code is independent of TypeScript’s type checking procedure. Still, we follow the same pragmatic approach and intentionally sacrifice strong type safety in the design of our static analysis algorithm in Section 8.5 to ensure high performance and reduce the number of false positives.
8.2 The TypeScript Declaration Language

TypeScript extends the JavaScript language with additional statements and expressions for interfacing with the type system. We here give an overview of the declaration language in TypeScript and introduce the notation and terminology that we use in the remainder of the paper.

Declarations in TypeScript can do one or both of the following: (1) contribute to the type namespace at compile time, and (2) have an effect at runtime. Declarations that have no effect at runtime are called ambient declarations. Declaration files consist exclusively of ambient declarations, so we will focus on the subset of TypeScript that may occur in such declarations.

An interface declaration declares a type and has no effect at runtime. For example, the interface below declares a type `IPoint` with members `x` and `y` of type `number`.

```typescript
interface IPoint {
    x: number;
    y: number;
}
```

TypeScript uses a structural type system; any object satisfies the `IPoint` type if it has properties `x` and `y` with numeric values. This stands in contrast to nominal type systems known from Java-like languages where an object only implements an interface if the class of the object is explicitly declared to implement the interface.

A class declaration declares a type and creates a constructor for the class at runtime. For example, consider the class below:

```typescript
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}
```

This declares a type `Point` but will also at runtime create a constructor function for the class and expose it in the variable `Point`, so that the class can be instantiated by an expression, such as, `new Point(4, 5)`. Class types are only satisfied by objects that are created using its constructor, so for instance, an object created by the expression `{x: 4, y: 5}` satisfies the `IPoint` type but not the `Point` type, while an object created by the expression `new Point(4, 5)` satisfies both `Point` and `IPoint`. This makes class types behave like the nominal types from Java-like languages. In TypeScript nomenclature, such types are called branded types.

The class declaration above has effects at runtime, and thus cannot occur in a declaration file. An ambient class declaration may be used instead, which instructs the TypeScript compiler to behave as if the class declaration had been there (for type checking purposes), but no code will be emitted for the declaration:

```typescript
declare class Point {
    x: number;
    y: number;
    constructor(x: number, y: number);
}
```
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

8.2.1 TypeScript Declaration Core

The TypeScript language has non-trivial name resolution and inheritance features, which we will not describe in this work. We instead present TypeScript Declaration Core that captures the expressiveness of TypeScript’s declaration language, after all type references have been resolved and inheritance hierarchies have been flattened.

We now describe the syntax and semantics of this core type system. We let \( S \) denote the set of Unicode strings for use as property names and string constants. A TypeScript declaration core consists of the following finite disjoint sets,

\[
\begin{align*}
T & \in N : \text{type names} \\
v & \in V : \text{type variables} \\
e & \in E : \text{enum type names}
\end{align*}
\]

along with a type environment, the name of the global object type, and a mapping from enum types to their members:

\[
\begin{align*}
\Sigma &: N \rightarrow \text{type-def} : \text{type environment} \\
G & \in N : \text{name of the global object type} \\
\Gamma &: E \rightarrow \mathcal{P}(S) : \text{enum members}
\end{align*}
\]

The type environment \( \Sigma \) maps type names to their definitions, e.g. \( \text{IPoint} \) would map to the type \( \{x:\text{number}; y:\text{number}\} \). In JavaScript, all global variables are actually properties of a special object known as the global object. We use \( G \) as the name of the global object type. The global object must satisfy the global object type, hence \( \Sigma(G) \) can be used to describe the types of global variables. Enum type names \( e \in E \) describe finite enumeration types; the enum member mapping \( \Gamma \) denotes what values belong to each enum type. Enum types will be discussed in more detail later in this section.

The syntax of types is given below:

\[
\begin{align*}
\text{type-def} & ::= \langle V \rangle \text{ obj} \\
\tau & \in \text{type} ::= \text{any} \mid \text{void} \mid \text{bool} \mid \text{num} \mid \text{str} \mid "S" \mid \text{obj} \mid N(\text{type}) \mid E \mid V \\
o & \in \text{obj} ::= \{ \text{prty} : \text{indexer} ; \text{fun-sig} ; \text{brand} \} \\
\rho & \in \text{prty} ::= \text{opt}? S : \text{type} \\
i & \in \text{indexer} ::= [\text{num} : \text{type}] \mid [\text{str} : \text{type}] \\
c & \in \text{fun-sig} ::= \text{new}? \text{vargs} \langle \text{type-parm}\rangle(\text{parm}) \Rightarrow \text{type} \\
p & \in \text{parm} ::= \text{opt}? \text{type} \\
\text{type-parm} & ::= V \text{ extends } \text{type} \\
b & \in \text{brand} ::= S
\end{align*}
\]

We will now informally describe the meaning of these types.

**Type Definitions** A type definition \( \langle \bar{v} \rangle \text{obj} \) defines a potentially generic type, with \( \bar{v} \) being the names of the type variables.

**Example** The \text{IPoint} interface corresponds to \( \Sigma(\text{IPoint}) = \langle \{x:\text{number}; y:\text{number}\} \rangle \).
Type References  A type of form $T\langle \tau \rangle$ denotes the instantiation of the type with name $T$ with type arguments $\tau$. Here, $T$ must be the name of a type definition that has the same number of type parameters. If zero type parameters are specified, this is simply a reference to the type $T$.

Objects  An object type consists of properties, indexers, function signatures, and brands. Each member restricts the set of values that satisfy the type. These meaning of these members are detailed below.

Properties  A property $f : \tau$ denotes that an object of this type must have a property $f$ with a value of type $\tau$. If the opt flag is specified, the property is optional. An optional property opt $f : \tau$ denotes that if the object has a property named $f$, then the value of that property must have type $\tau$.

Indexers  An indexer [str : $\tau$] denotes that all enumerable properties of the object must have a value of type $\tau$. Such indexers are used for describing the types of objects that are used as dictionaries. Enumerability is a boolean flag set on each property, indicating if it should be seen by reflective operators, such as JavaScript’s for-in loop. Properties are generally enumerable by default, but certain natively defined properties are non-enumerable so that they do not interfere with dictionary-like objects.

An indexer [num : $\tau$] denotes that all properties whose name is a valid array index (i.e. "0", "1", "2", etc.) must satisfy $\tau$. JavaScript treats array indices as properties whose names are number strings, so number indexers can be used to describe the types of arrays and array-like objects.

Example  The declaration file lib.d.ts, which the TypeScript compiler uses as a model for JavaScript’s native API, uses number indexers to describe the type of arrays (TypeScript uses a slightly different syntax for indexers than our core language):

```typescript
interface Array<T> { ... 
  [n: number]: T; 
} 
```

Indexers are not only used for actual arrays. The declaration file for jQuery shows that a jQuery object can be seen as an array of HTMLElements:

```typescript
interface JQuery { ... 
  [n: number]: HTMLElement; 
} 
```

Function Signatures  A function signature in an object type denotes that the object must be callable as a function; such an object is typically called a function. In its simplest form, a signature $(\tau_1, ..., \tau_n) \Rightarrow \tau$ specifies that when the function is invoked with $n$ arguments satisfying types $\tau_1, ..., \tau_n$, a value of type $\tau$ should be returned. If the new flag is present, the signature only applies to constructor calls, otherwise, it only applies to non-constructor calls. If the vargs flag is present, the signature additionally applies to any call with more than $n$ arguments where the extra arguments satisfy the type $\tau_n$. The vargs flag cannot be used if $n = 0$. If the last $k$ parameters are annotated with the opt flag, the signature additionally applies to calls with $n - k$ arguments. It is invalid to use the
opt flag for a parameter unless all following parameters also use it. A function signature is polymorphic if it declares one or more type parameters; in this case the function must satisfy all valid instantiations of the function signature. An instantiation is valid if the types substituting a type variable satisfies its bound.

**Brands** An object type with a brand \( b \) is called a *branded type*. Brands serve to emulate nominal types in TypeScript’s structural type system, so that otherwise equivalent types are distinguishable. In TypeScript, classes are branded types while interfaces are unbranded. In our core type system, a brand is an access path pointing to the prototype object associated with a class. An access path is a sequence of property names, for example, the path \( \text{foo.bar} \) points to the value of the \( \text{bar} \) property of the object in the global variable \( \text{foo} \). For an object to satisfy a brand \( b \), this object must have the object pointed to by \( b \) in its prototype chain.

Given a class declaration `declare class C {}`, the branded type \( \Sigma(\text{C}) = \langle \rangle \{ \text{C.prototype} \} \) would be generated. This interpretation of branded types is based on how constructor calls work in JavaScript: when an object \( o \) is created by an expression of form `new F(...)`, the internal prototype link of \( o \) is initialized to the object referred to by \( \text{F.prototype} \). Thus, the prototype chain indicates what constructor was used to create a given object.

**Enums** An enum type of name \( e \) is satisfied by the values pointed to by the access paths in \( \Gamma(e) \).

**Example** The enum declaration below declares an enum type of name `FontStyle` ∈ \( E \).

```typescript
52 declare enum FontStyle {
53   Normal, Bold, Italic
54 }
```

The declaration also introduces a global variable `FontStyle` with members `Normal`, `Bold`, `Italic`, all of type `FontStyle`. The values for these three members are not specified in the type and can be chosen arbitrarily by the implementation. The specific values used by the implementation of this enum can be found by inspecting the access paths `FontStyle.Normal`, `FontStyle.Bold`, and `FontStyle.Italic`. For instance, here are two valid JavaScript implementations of the enum type:

```typescript
55 var FontStyle = // implementation 1
56   { Normal: 0, Bold: 1, Italic: 2 }
57 var FontStyle = // implementation 2
58   { Normal: "normal", Bold: "bold",
59     Italic: "italic" }
```

In both cases, we have \( \Gamma(\text{FontStyle}) = \{ \text{FontStyle.Normal, FontStyle.Bold, FontStyle.Italic} \} \), but the set of values satisfied by the enum type depends on the implementation being checked against.

### 8.3 Library Initialization

In this section we discuss what it means to initialize a library, the assumptions we make about the library initialization process, and the notion of a snapshot of the runtime state.
8.3. Library Initialization

Loading a library in JavaScript amounts to executing the library’s top-level code. In order to expose its functionality to clients, this code must introduce new global variables and/or add new properties to existing objects. Although not strongly enforced in any way, it is typically the case that a library is initialized and its API is ready to be used when the end of its top-level code is reached. In principle, it is possible for a library to postpone parts of its initialization until the client calls some initialization function or an event is fired from the browser, but this is seldom seen in practice. We exploit this common case using the following assumption:

**Initialization assumption:** The library’s public API has been established at the end of its top-level code.

Another way to phrase the initialization assumption is that the declaration file is intended to describe the state of the heap after executing the top-level code. The experiments discussed in Section 8.6 confirm that this assumption only fails in rare cases.

JavaScript uses an event-based single-threaded programming model. This means that it is not possible for a JavaScript library to read any input at the top-level, neither through the UI nor using AJAX calls. Though it may initiate an AJAX call, it cannot react to the response until after the top-level code has finished and the event loop has resumed. Thereby we can execute the top-level code in isolation without needing to simulate inputs from the environment.

JavaScript has multiple execution platforms; notably, there are at least five major browsers as well as the server-side platform node.js. These platforms provide incompatible native APIs, which means it is possible, and often necessary, for a library to detect what platform it is running on. Although the top-level code cannot read any external input, the platform itself can be seen as an input. We could take a snapshot using every version of every major platform, but this is somewhat impractical. Fortunately, libraries go to great lengths to provide platform-agnostic APIs. As long as the API is completely platform-agnostic, any declaration bug we can detect on one platform should be detectable on all supported platforms. In practice, it should therefore suffice to run the library on just one platform. This leads to our second assumption:

**Platform assumption:** Executing the library on a single platform (as opposed to multiple platforms) is sufficient for the purpose of uncovering bugs in the declaration file.

Note that our goal is to find errors in the TypeScript declarations, not in the JavaScript implementation of the library. If we were looking for bugs in the library implementation, the platform assumption might not be as realistic. As with the previous assumption, experiments will be discussed in the evaluation section, which confirm that this assumption is valid in practice.

While some libraries are completely portable, there are also some that are specific to the browser platforms, and some are specific to the node.js platform. Our implementation supports two platforms: the phantom.js platform based on WebKit (the engine used in Chrome and Safari) and the node.js platform. Most JavaScript libraries support at least one of these two platforms.

---

141
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

We now describe the structure extracted from the program state at runtime. The call stack is known to be empty when extracting the snapshot, because we are at the end of the top-level code, so this is effectively a snapshot of the heap.

We use boldface fonts for symbols related to the snapshot. A snapshot consists of a finite set of object names \( O \),

\[
k \in O \quad : \quad \text{object names}
\]

along with a name for the global object \( g \), and a store \( s \),

\[
g \in O \quad \quad \text{(global object)}
\]
\[
s : O \to \text{obj-def} \quad \text{(store)}
\]

where the definition of objects and values are as follows:

\[
x, y \in \text{value ::= true | false | num(\mathbb{D}) | str(S) | null | undef | obj(O)}
\]
\[
\text{obj-def ::= \{ properties: \text{prty};}
\]
\[
\text{prototype: value;}
\]
\[
\text{function: fun;}
\]
\[
\text{env: value } \}
\]
\[
\text{fun ::= user(U) | native(S) | none}
\]
\[
\text{p \in \text{prty ::= \{ name : S}
\]
\[
\text{enumerable : true | false}
\]
\[
\text{value : value;}
\]
\[
\text{getter : value;}
\]
\[
\text{setter : value } \}
\]

Here, \( \mathbb{D} \) is the set of 64-bit floating point values, and \( U \) is a set of identifiers for the functions in the source code from which the snapshot was taken.

An object has a list, \( \text{properties} \), representing its ordinary properties. A property can have a \( \text{value} \) field, or it can have a \( \text{getter/setter} \) pair denoting functions that are invoked when the property is read from or written to, respectively. For properties with a non-null \( \text{getter} \) or \( \text{setter} \), the \( \text{value} \) field is always \( \text{null} \).

An object’s \( \text{prototype} \) field represents JavaScript’s internal prototype link; the object inherits the properties of its prototype object. This field is set to \( \text{null} \) for objects without a prototype.

The \( \text{function} \) field indicates whether the object is a user-defined function, a native function, or not a function. Variables captured in a closure are represented as properties of environment objects. For user-defined functions, the \( \text{env} \) field refers to the environment object that was active when the function was created, thus binding its free variables. For environment objects, the \( \text{env} \) field points to the enclosing environment object, or the global object. Environment objects cannot be addressed directly in JavaScript (we ignore \( \text{with} \) statements), and thus can only be pointed at by the \( \text{env} \) field. Environment objects do not have a prototype.

8.4 Type Checking of Heap Snapshots

We now show how a value from the heap snapshot can be checked against a type from the declaration file. In this section we ignore function signatures (i.e., they are assumed to be
satisfied) and getters and setters (i.e. they are assumed to satisfy any type), which makes this type checking problem decidable.

We introduce typing judgements with five different forms:

\[ \approx \subseteq \text{value} \times \text{type} \cup \text{prty} \times \text{type} \cup \text{O} \times \text{prty} \cup \text{O} \times \text{indexer} \cup \text{O} \times \text{brand} \]

The judgement of form \( x \approx \tau \) means the value \( x \) satisfies the type \( \tau \). A judgement \( p \approx \tau \) means the concrete property \( p \) has a value satisfying \( \tau \) (or it has a getter/setter). A judgement \( k \approx \rho \) means the object with name \( k \) satisfies the property type \( \rho \); likewise, \( k \approx i \) means the object with name \( k \) satisfies the indexer type \( i \), and \( k \approx b \) means the object with name \( k \) satisfies the brand \( b \).

We use the rules in Figure 8.2 to derive typing judgements. When a question mark occurs after a flag in the inference rules it means the rule can be instantiated with or without that flag in place. Type checking is initiated by checking if the global object satisfies its type, i.e., we attempt to derive the typing judgement \( \text{obj}(g) \approx G(\tau) \) by goal-directed application of these rules.

The inference rules are cyclic, that is, the derivation of a typing judgement may loop around and depend on itself. When we encounter a cycle, which may happen due to cyclic structures in the heap snapshot and the corresponding types, we do not recursively check the judgement again, but coinductively assume that the nested occurrence of the judgement holds.

Several auxiliary functions are used in Figure 8.2; these are described in the following paragraphs.

The property lookup function \( \mathcal{L} : \text{O} \times S \to \text{prty} \cup \{\text{nil}\} \) maps an object name and a property name to the corresponding property definition (if any) while taking into account inheritance from prototypes:

\[
\mathcal{L}(k, f) = \begin{cases} 
  p & \text{if } s(k) \text{ has a property } p \text{ named } f \\
  \mathcal{L}(k', f) & \text{else if } s(k).\text{prototype} = \text{obj}(k') \\
  \text{nil} & \text{otherwise}
\end{cases}
\]
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

This is well-defined since prototype chains cannot be cyclic.

We use the notation \( L(k, \ast) \) to denote the finite set of all properties available on the object with name \( k \). The phrase “has a property” used in the above definition means the properties sequence of the object should contain a property with the given name.

The path lookup function \( L^* \) operates on access paths instead of single property names. It returns a value instead of a property, and defaults to nil if the path cannot be resolved:

\[
L^*: O \times S^* \rightarrow \text{value} \cup \{\text{nil}\}
\]

\[
L^*(k, f) = \begin{cases} 
\text{obj}(k) & \text{if } f = \varepsilon \\
L^*(k', f') & \text{if } f = f_1 f' \land L(k, f_1) = p \land p\text{.value} = \text{obj}(k') \\
\text{nil} & \text{otherwise}
\end{cases}
\]

The ToObj function, ToObj : value \rightarrow value, converts primitive values to objects:

\[
\text{ToObj}(x) = \begin{cases} 
\text{obj}(k) & \text{if } x = \text{obj}(k) \\
L^*(g, \text{Number.prototype}) & \text{if } x = \text{num}(z) \\
L^*(g, \text{String.prototype}) & \text{if } x = \text{str}(s) \\
L^*(g, \text{Boolean.prototype}) & \text{if } x = \text{true} \\
L^*(g, \text{Boolean.prototype}) & \text{if } x = \text{false} \\
\text{null} & \text{otherwise}
\end{cases}
\]

A path such as \( \text{Number.prototype} \) should be read as the path consisting of two components: the string \( \text{Number} \) followed by the string \( \text{prototype} \). Note that the property named “prototype” should not be confused with the internal prototype link.

In JavaScript, a primitive value is automatically coerced to an object when it is used as an object. For instance, when evaluating the expression "foo".toUpperCase() in JavaScript, the primitive string value "foo" is first coerced to a string object, and this object inherits the toUpperCase property from the String.prototype object. To allow such expressions to be typed, TypeScript permits an automatic type conversion from primitive types to their corresponding object types. The ToObj function is used in the [obj] rule to mimic this behavior.

The Protos function, Protos : O \rightarrow \mathcal{P}(O), maps the name of an object to the set of objects on its prototype chain:

\[
\text{Protos}(k) = \begin{cases} 
\text{Protos}(k') \cup \{k\} & \text{if } s(k).\text{prototype} = \text{obj}(k') \\
\{k\} & \text{otherwise}
\end{cases}
\]

The substitution operator \( o[\tau \rightarrow \tau] \) used in the [ref] rule produces a copy of \( o \) where for all \( i \), all occurrences of the type variable \( v_i \) have been replaced by the type \( \tau_i \).

We will now discuss some of the typing rules in more detail.

**Primitives** The rules for the primitive types, any, num, bool, str, and "s" are straightforward. Following the TypeScript specification, null satisfies any type, and undef satisfies the void type.
8.4. Type Checking of Heap Snapshots

**Enums**  As mentioned in Section 8.2, $\Gamma(e)$ denotes a set of access paths pointing to the members of enum $e$. The [enum] rule uses the path lookup function $L^*$ to resolve each access path and check if the value is found at any of them.

**Example**  Below is an enum declaration and an implementation in JavaScript:

```javascript
60 // declaration file:
61 declare enum E { X, Y }
62 // JavaScript file:
63 var E = { X: "foo", Y: "bar" }
```

To check that the string value "bar" satisfies the enum type $E$, we can apply the [enum] rule with $f = E.Y \in \Gamma(E)$, and find that $L^*(g, E.Y) = "bar"$.

**References**  The [ref] rule looks up a reference in the type environment $\Sigma$, instantiates the type with the given type arguments (if any), and recursively checks the value against the resolved type.

**Example**  Below is a generic type $Pair<T>$ and a variable whose type is an instantiation of this type:

```javascript
64 // declaration file:
65 interface Pair<T> { fst: T; snd: T; }
66 declare var x : Pair<number>;
67 // JavaScript file:
68 var x = { fst: 42, snd: 24 };
```

If we let $x$ denote the concrete object created on line 68, the type checker will attempt to derive $x \sim Pair<num>$. The [ref] rule will then be applied, after which the type checker will recursively attempt to derive $x \sim \{ fst : num; snd : num \}$.

Note that TypeScript requires that a generic type does not contain a reference to itself with type arguments wherein one of its type parameters have been wrapped in a bigger type. For instance, the following type declaration violates this rule:

```javascript
69 interface F<T> { x: F<T[], T[]> }
```

This means only regular types can be expressed. Without this condition, termination of our type checking procedure would not be guaranteed.

**Objects**  The [obj] rule coerces the value to an object, and checks separately that the resulting object (if any) satisfies each property, indexer, and brand member of the type.

**Properties**  The [prty] and [prty-nil] rules use the lookup function $L$ to find the property with the given name (if any). The [prty-nil] rule allows optional property members to be satisfied when the concrete property does not exist. If the property exists, the [prty] rule will check its type; in most cases this rule will be followed up by an application of the [p-value] rule. In case the property is defined by a getter and/or setter, the [p-getter] rule automatically accepts the typing judgement.

**Indexers**  The [str-idx] rule checks that every enumerable property satisfies the type, and [num-idx] checks that every array entry (i.e. property whose name is a number string) satisfies the type. As with the [prty] rule, these rules are typically followed up by an application of [p-value].
Brands  The [brand] and [brand\_nil] rules use the path lookup function $L^*$ to find the brand’s associated prototype object, and then checks that this object exists on the prototype chain. The [brand\_nil] rule exists to avoid duplicate errors in case a class appears to be missing, since this would otherwise be reported as a missing class constructor and as missing brands for every instance of the class.

Functions  The [obj] rule as shown in Figure 8.2 ignores the function signatures of the type being checked against because checking these requires static analysis of the function body. We collect all such objects and corresponding function signatures in a list that will be checked in the static analysis phase described in Section 8.5.

8.4.1 Relationship to TypeScript

These type checking rules formalize the relation between values and types in TypeScript, which is only described implicitly and informally in the TypeScript language specification. We have strived toward a faithful formalization, however, in two situations we have decided to deviate from the language specification in order to match common uses of TypeScript types in declaration files.

First, the specification states that enum types are assignable to and from the number type, but different enum types are not assignable to each other (assignability is not transitive in TypeScript). While this may be appropriate for code written entirely in TypeScript, it does not reflect how JavaScript libraries work, where string values are often used as enum-like values. Despite the tight coupling to numbers, we see declaration files use enums to describe finite enumerations of string values. Thus, we developed the access-path based interpretation of enums to allow for more liberal use of the enum construct when checking against a JavaScript library.

Second, the TypeScript specification treats branded types purely as a compile-time concept. Since they are only used for class types, there is a simple runtime interpretation of brands, which we decided to include as reflected by the [brand] rule.

8.5 Static Analysis of Library Functions

The type checking algorithm presented in the preceding section is designed to detect mismatches between the object structure in the snapshot and the declarations, however, it does not check the function code. In this section we describe a static analysis for determining if a given function implemented in JavaScript satisfies a given function signature. The problem is undecidable, so any analysis must occasionally reject a correct function implementation, or conversely, accept an incorrect one. Following the philosophy of optional types, in which usability is valued higher than type safety, we aim toward the latter kind of analysis so as not to disturb the user with false warnings.

Figure 8.3 shows an example of a bug that was found using the algorithm presented throughout this section. On line 97, a function bundle is declared to return a BundleLayout that, as per line 99, itself must be a function taking a single GraphLink[] argument and returning a GraphNode[]. The implementation, however, returns a two-dimensional array of GraphNodes; the return type on line 99 should thus have been GraphNode[][].

Our starting point is the collection of pairs of function objects and corresponding function signatures that was produced in connection with the [obj] rule, as explained in
8.5. Static Analysis of Library Functions

Fragment of d3.js (library implementation)

```javascript
80 d3.layout.bundle = function() {
  return function(links) {
    var paths = []
    for (var i=0; i<links.length; ++i) {
      paths.push(d3_layout_bundlePath(links[i]))
    }
    return paths;
  };
};
function d3_layout_bundlePath(link) {
  var start = link.source
  var end = link.target
  var lca = d3_layout_bundleLCA(start, end)
  var points = [ start ]
  while (start !== lca) {
    start = start.parent
    points.push(start)
  }
  var k = points.length
  while (end !== lca) {
    points.splice(k, 0, end)
    end = end.parent
  }
  return points
}
function d3_layout_bundleLCA(a,b) {...}
```

Fragment of d3.d.ts (declaration file)

```typescript
declare module d3.layout {
  function bundle(): BundleLayout
  interface BundleLayout{
    (links: GraphLink[]): GraphNode[]
  }
  interface GraphLink {
    source: GraphNode
    target: GraphNode
  }
  interface GraphNode {
    parent: GraphNode
    /* some properties omitted ... */
  };
}
```

Figure 8.3: Example of an incorrect function signature found in d3.d.ts (slightly modified for readability).
Section 8.4 For each of these pairs, we now analyze the function body and check that it is compatible with the function signature. For each program expression, the analysis will attempt to compute a type that over-approximates the set of values that may flow into the given expression. Once an approximate type has been found for every expression, we determine if the type inferred for the function’s possible return values is assignment compatible with the function signature’s declared return type. We will discuss the notion of assignment compatibility later, but for now, it can be thought of as the types having a nonempty intersection. If the types are not assignment compatible, i.e. their intersection is empty, then any value the function might return will violate the declared return type, indicating a clear mismatch between the declaration file and the library implementation. If the types are assignment compatible, on the other hand, we view the function signature as satisfied.

The analysis of each function can be seen as a two-step process. In the first step, a set of constraints is generated by a single traversal over the AST of the library code and the heap snapshot. The heap snapshot is used in the process, for example to resolve global variables. In the second step, the constraints are solved by finding a least fixed-point of some closure rules. We will introduce various types of constraints and discuss the most relevant AST constructs in the following subsections.

The analysis will involve all functions that might transitively be called from the function being checked. As an example, the implementation of bundle on lines 70–78 in Figure 8.3 uses the function d3_layout_bundlePath for which we do not have a function signature, so checking bundle also involves the code in d3_layout_bundlePath.

8.5.1 Atoms and Unification

To reason about the types of program expressions, we introduce a new type, called an atom:

\[
\text{type ::= \ldots | atom}
\]

There is an atom for every expression in the AST and for every object in the heap snapshot, and some additional atoms, which we will introduce as needed:

\[
a \in \text{atom ::= ast-node | O | \ldots}
\]

These atoms can be seen as type variables, representing the type of a program expression (which is unknown due to lack of type annotations in the source code) or the type of a heap object (which is known at the time of the snapshot, but may later be modified by the program code, hence should also be seen as unknown).

We define a term to be an atom or an atom combined with a field name. A field can either be the name of a property (i.e. a string) or the special env field name that we use to model the chain of environment objects:

\[
t \in \text{term ::= atom | atom \diamond field}
\]

\[
f \in \text{field ::= S | env}
\]

Like atoms, terms can be thought of as type variables. Intuitively, a compound term \( a \diamond f \) acts as placeholder for the type one gets by looking up the property \( f \) on \( a \). This lets us address the type of \( a \)'s properties at a time where the type of \( a \) is still unknown. In an
analogy to points-to analysis, the compound term \( a \diamond f \) can be thought of as a placeholder for whatever is in the points-to set for \( a.f \).

Our analysis computes an equivalence relation \( \equiv \subseteq \text{term} \times \text{term} \) of terms that are deemed to have the same type. When \( t_1 \equiv t_2 \), we say that \( t_1 \) and \( t_2 \) have been unified. The analysis assigns a single type to each equivalence class in \( \equiv \), so once terms have been unified, they can effectively be thought of as being the same term.

In addition to being an equivalence relation, \( \equiv \) is closed under the [prty] rule, which states that once two atoms have been unified, all their fields must also be pointwise unified. In summary, \( \equiv \) is closed under following four rules (the first three ones being the rules for equivalence relations).

\[
\begin{align*}
\frac{t_1 \equiv t_2, \ t_2 \equiv t_3}{t_1 \equiv t_3} & \quad \text{[trans]} \\
\frac{t \equiv t}{t_1 \equiv t_2} & \quad \text{[refl]} \\
\frac{t_1 \equiv t_2}{t_2 \equiv t_1} & \quad \text{[sym]} \\
\frac{a_1 \equiv a_2}{a_1 \diamond f \equiv a_2 \diamond f} & \quad \text{[prty]}
\end{align*}
\]

To gain some intuition about the meaning of this relation, one can think of it as a points-to graph in which every equivalence class of \( \equiv \) is a vertex, and there is an edge labelled \( f \) from the class of an atom \( a \) to the class of \( a \diamond f \). The following graph visualizes the state of \( \equiv \) generated from d3_layout_bundlePath from Figure 8.3 (if ignoring function calls), where the labels inside each vertex indicate which atoms reside in the class:

![Graph visualization of equivalence classes](image)

We use a union-find data structure to represent the equivalence classes of \( \equiv \). Every root node in the data structure contains its outgoing edges in a hash map. When two nodes are unified, their outgoing edges are recursively unified as well. More details of this procedure can be found in our previous work on JavaScript analysis for refactoring [24], which follows a related approach with a simpler constraint system.

The \( \equiv \) relation allows us to handle three basic forms of program expressions:

**Local Variables** We unify all references to the same variable, thus capturing all data flow in and out of the variable. Closure variables will be discussed in Section 8.5.4.

**Properties** An expression \( e \) of form \( e_1.f \) is translated into the constraint \( e \equiv e_1 \diamond f \). By the transitivity and the [prty] rule, \( e \) will become unified with all other occurrences of \( f \) on a similarly typed expression.

**Assignment** An expression \( e \) of form \( e_1 = e_2 \) is translated into the constraints \( e \equiv e_1 \) and \( e \equiv e_2 \).
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

8.5.2 Types

For every equivalence class in \( \equiv \) we compute a set of types:

\[
\text{Types} : \text{term} \xrightarrow{\equiv} \mathcal{P}(\text{type})
\]

We use the \( \equiv \) symbol to denote functions that are congruent with \( \equiv \), so that whenever \( a_1 \equiv a_2 \), we must have \( \text{Types}(a_1) = \text{Types}(a_2) \). Our implementation maintains one set of types per root in the union-find data structure and merges the sets when two roots are unified.

The primary purpose of these types is to encode the types of the parameters from the function signature being checked. A type \( \tau \) associated with a term \( t \) can be seen as a bound on the type of \( t \). In an analogy to dataflow analysis, one can also think of \( \tau \) as a set of values that may flow into \( t \).

Types are propagated to other terms by the following rule, where \( \tau.f \) indicates the lookup of property \( f \) on the type \( \tau \):

\[
\frac{\tau \in \text{Types}(a)}{\tau.f \in \text{Types}(a \circ f)} \quad \text{[type-field]}
\]

We unify two atoms if the type of one becomes bound by the other:

\[
\frac{a_1 \in \text{Types}(a_2)}{a_1 \equiv a_2} \quad \text{[type-same]}
\]

An example is shown in Section 8.5.6. We also introduce primitive values as types, so we can use types for a kind of constant propagation (note that we never use \( \text{obj} \) values as types):

\[
\text{type ::= } \ldots \mid \text{value}
\]

This lets us translate two more forms of basic expressions:

**Literals** A constant expression \( e \) with primitive value \( x \) is translated into the constraint \( x \in \text{Types}(e) \). Similarly, if \( e \) is a simple object literal \( \{ \} \), the constraint \( \{ \} \in \text{Types}(e) \) is generated.

**Array Expressions** For an expression \( e \) of form \( [e_1, \ldots, e_n] \), we generate a new atom \( a \) to represent the members of the array being created. We then add the constraints \( \text{Array}(a) \in \text{Types}(e) \) and \( a \equiv e_i \) for every \( i \).

**Arithmetic Operators** Arithmetic operators, such as \( * \) and \( - \), are assumed to return values of type \( \text{num} \), while the + operator is assumed to return a value of type \( \text{str} \) or \( \text{num} \).

We can now continue the previous example with types on the vertices (again, we show the function in isolation without interprocedural information):
The atom $a_1$ represents the members of the points array, and $c_1$ is short for the function signature $\text{vargs}(a_1) \Rightarrow \text{num}$, which is the type for the push method on arrays of type $\text{Array}(a_1)$. Likewise, $c_2$ and $c_3$ represent the two signatures of the overloaded splice method on arrays. The [type-field] rule propagates types along the four outgoing edges from the node with type $\text{Array}(a_1)$, for example, the type num along the length edge.

8.5.3 Dynamic Property Accesses

A JavaScript expression of form $e_1[e_2]$ is called a dynamic property access. When evaluated, the result of evaluating the expression $e_2$ is coerced to a string, which is then looked up as a property on the object obtained from $e_1$. This type of expression is also used to access the elements of an array. To handle this construct in the library code, we generate constraints of form $(a_{obj}, a_{prty}, a_{res}) \in \text{DynAccess}$ where

$$\text{DynAccess} \subseteq \text{atom} \times \text{atom} \times \text{atom}$$

We use the types of the property name to determine what properties might be addressed. For concrete primitive values we convert the value to a string and use that as a property name (where $\text{ToString}$ models coercion of primitive values but ignores objects):

$$\frac{(a_{obj}, a_{prty}, a_{res}) \in \text{DynAccess}, \ x \in \text{Types}(a_{prty})}{a_{obj} \diamond \text{ToString}(x) \equiv a_{res}} \quad \text{[dyn-val]}$$

If the property is a computed number, we merge all numeric properties of the object:

$$\frac{(a_{obj}, a_{prty}, a_{res}) \in \text{DynAccess}, \ num \in \text{Types}(a_{prty})}{\forall i \in \mathbb{N} : a_{obj} \diamond i \equiv a_{res}} \quad \text{[dyn-num]}$$

If the property is a computed string, we merge all properties of the object:

$$\frac{(a_{obj}, a_{prty}, a_{res}) \in \text{DynAccess}, \ str \in \text{Types}(a_{prty})}{\forall s \in S : a_{obj} \diamond s \equiv a_{res}} \quad \text{[dyn-str]}$$

The last two rules are implemented by keeping two flags on every root in the union-find data structure indicating whether its numeric properties, or all its properties, respectively, should be merged.

8.5.4 Function Calls

Functions are first-class objects in JavaScript. To reason about the possible targets of a function call inside the library we therefore track a set of functions for each atom:

$$\text{Funcs : } \text{atom} \rightarrow \mathcal{P}(\text{fun})$$

The set $\text{Funcs}(a)$ contains the user-defined and native functions that may flow into $a$.

We use the function FunDef to map function names $U$ to the atoms representing the input and output of the function:

$$\text{FunDef : } U \rightarrow \text{atom} \times \text{atom} \times \text{atom} \times \text{atom}$$
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

The first three atoms represent inputs to the function: the function instance itself, the this argument, and the arguments object that contains the actual argument in its properties. The last atom represents the function’s return value.

Similarly, we maintain a set Calls, also consisting of tuples of four atoms:

\[
\text{Calls} \subseteq \text{fun. instance} \times \text{this arg} \times \text{arg tuple} \times \text{return atom}
\]

The first three atoms represent the function instance, the this argument, and the arguments object passed to the called function, and the last atom denotes where to put the value returned by the function. Each syntactic occurrence of a function or a function call gives rise to an element in Funcs or Calls, respectively. Constructor calls (i.e. calls using the new keyword) will be discussed separately in Section 8.5.8.

For every potential call target at a function call, we unify the corresponding atoms:

\[
\frac{(a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \in \text{Calls}, \text{user}(u) \in \text{Funcs}(a_{\text{fun}})}{(a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \equiv \text{FunDef}(u)[\text{call}]}
\]

The symbol \(\equiv\) should be read as the pointwise application of \(=\), for example, the arguments of the call are unified with the arguments from the function definition. Note that our treatment of function calls is context insensitive.

Closure variables (i.e. variables defined in an outer scope) are accessible to the called function through the function instance argument, whose env field points to its enclosing environment object. Each environment object is represented by an atom whose env field points to its own enclosing environment, ending with the global object.

As an example, the diagram below shows some of the constraints generated for the function on lines 71–77. The call to paths.push is omitted for brevity. The DA node represents an entry in DynAccess generated for the links[i] expression, and the Call node represents the entry in Calls that was generated for the call to d3_layout_bundlePath (though we omit its associated this and return atoms):

8.5.5 Entry Calls

The analysis maintains a set of entry calls that denote places where the library code gets invoked by an external caller (i.e. from the client code or the native environment). An entry call consists of two atoms and a function signature:

\[
\text{EntryCalls} \subseteq \text{fun} \times \text{this arg} \times \text{fun-sig}
\]
8.5. Static Analysis of Library Functions

The first atom denotes the function instance being invoked, and the other denotes the this argument used in the call. The function signature associated with an entry call provides type information for the parameters.

Initially, a single entry call exists to the function being checked against a function signature. This serves as the starting point of the analysis, but more entry calls may be added during the analysis, as will be discussed in Section 8.5.6.

The following rule describes how entry calls are handled (for simplicity, ignoring variadic functions, optional parameters, and constructor calls):

\[
(a_{\text{fun}}, a_{\text{this}}, (\tau) \Rightarrow \tau') \in \text{EntryCalls}, \quad \text{user}(u) \in \text{Funcs}(a_{\text{fun}}), \quad \text{FunDef}(u) = (a'_{\text{fun}}, a'_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \quad \text{[entry]}
\]

In the previous example, the [entry] rule was responsible for adding the Array⟨GraphLink⟩ type.

Note that calls between two library functions are handled as explained in Section 8.5.4, not as entry calls, even if the called function has a signature in the declaration file.

8.5.6 Exit Calls

When we encounter a function call to a native function or a function that was provided through an argument, we handle the call using the function signature for the function. We refer to these as exit calls as they target functions that are defined outside the library code. For brevity, we give only an informal description of our treatment of exit calls.

When resolving a call \((a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \in \text{Calls}\) where \(\text{Types}(a_{\text{fun}})\) contains an object type with a function signature \(c\), we check if the arguments might satisfy the parameter types of \(c\) by an assignment compatibility check, and if the check succeeds, the return type of the function signature is added to \(\text{Types}(a_{\text{ret}})\). The check helps handle overloaded functions by filtering out signatures that are not relevant for a given call site.

To model callbacks made from the external function, whenever an atom \(a\) is compared to an object type with a function signature \(c\) and there is a function \(\text{user}(u) \in \text{Funcs}(a)\), we register a new entry call to \(a\) with the function signature \(c\).

If the function signature being invoked is polymorphic, the compatibility check against the parameter types also serves to instantiate the type variables in the function signature. Whenever a type variable \(V\) is compared to an atom \(a\) in the compatibility check, \(a\) is registered as a bound on \(V\). If the entire compatibility check succeeds then all atoms that bind the same type variable are unified.

The standard library is modeled using the types from the TypeScript compiler’s lib.d.ts file, except for call and apply which are given special treatment by the analysis.

The example below demonstrates how exit calls can handle array manipulation:

```javascript
110 var array = [];
111 var w = 42;
112 array.push(w);
113 var z = array[0];
```
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

Below is the state of the constraint system before the call to `push` has been resolved (here we use :: to separate atom names from their types):

The assignment on line 113 has caused `array` ⋄ θ to be unified with z. The [type-field] rule has then propagated a₁ as a type to z, and [type-same] then caused a₁ and z to become unified.

The call to `push` is an exit call to a function with signature (a₁) ⇒ void. The atom a₁ occurs as parameter type in this signature, and when it gets matched against the argument w the two atoms are unified. The dotted box above shows this unification.

This demonstrates how the value 42 propagated from w through the array into the variable z. Note that such destructive update of the arguments only happens when an atom occurs as part of a parameter type. This feature is our only way of modeling the side effects of exit calls.

8.5.7 Prototypes

One possible way to handle JavaScript’s prototype-based inheritance mechanism is to unify two atoms a₁ and a₂ if one inherits from the other. In practice, however, almost all objects would then become unified with `Object.prototype`, thereby destroying precision entirely. We instead represent inheritance by maintaining a set of prototypes for each atom:

\[ \text{Protos} : \text{atom} \equiv P(\text{atom}) \]

Intuitively, the set Protos(a) indicates what objects may reside on the prototype chain of a. The following rule ensures that properties that exist on a₁ will be shared with a₂:

\[ \frac{a₁ \in \text{Protos}(a₂), \text{NonEmpty}(a₁ \circ f) \quad \text{[inherit]} }{a₁ \circ f \equiv a₂ \circ f} \]

The predicate NonEmpty(t) holds if the current type associated with t is nonempty; we use it to check the existence of fields in the prototype.

\[ \text{NonEmpty}(t) = (\text{Types}(t) \neq \emptyset \lor \exists a. t \equiv a) \]

Without this check, by transitivity all instances of the same prototype would share all properties, effectively making inheritance the same as unification.

8.5.8 Constructor Calls

Like regular functions calls, constructor calls are recorded in a set of tuples:

\[ \text{NewCalls} \subseteq \text{fun. instance} \times \text{arg tuple} \times \text{return} \]

\[ \text{NewCalls} \subseteq \text{atom} \times \text{atom} \times \text{atom} \]
Constructor calls have no explicit this argument; instead, a newly created object is passed as this and is (in most cases) returned by the call.

\[
(a_{\text{fun}}, a_{\text{arg}}, a_{\text{ret}}) \in \text{NewCalls}, \ \text{user}(u) \in \text{Funcs}(a_{\text{fun}}) \implies \text{FunDef}(u) \quad [\text{new}]
\]

The prototype of the newly created object is the object referred to by the prototype property of the function being called. For every constructor call we generate an atom \(a_{\text{proto}}\) to represent the prototype of the newly created object, and add this as a prototype of the created object:

\[
a_{\text{fun}} \odot \text{prototype} \equiv a_{\text{proto}}, \quad a_{\text{proto}} \in \text{Protos}(a_{\text{ret}})
\]

As an example, consider this program fragment:

```javascript
114  function Foo(x) { /* ... */ }
115  Foo.prototype.bar = function() { /* ... */ }
116  var foo = new Foo();
117  foo.bar();
```

Below is the constraint system before applying the [inherit] rule (not showing the Calls constraint):

The dotted box on the right shows the two atoms that will be unified when applying the [inherit] rule. The topmost of these corresponds to the function created on line \(114\) and the bottommost one corresponds to the bar property mentioned on line \(117\).

### 8.5.9 Generating Constraints from the Snapshot

We now briefly discuss how information from the heap snapshot is embedded in the constraint system. Recall that there exists an atom for every object in the heap. The following rule connects pointers between objects:

\[
p \in L(k,*) , \ p.\text{value} = \text{obj}(k') \implies \ k \odot p.\text{name} \equiv k' \quad [\text{obj-value}]
\]

For getters or setters, we generate a call to the getter/setter and unify the result/argument with the term representing the property. Thereby the effects of the getters and setters will be taken into account whenever the property is referenced, without needing to generate calls at every use site:

\[
p \in L(k,*) , \ p.\text{get} = \text{obj}(k') \implies \ (k', k, a_{\text{arg}}, a_{\text{ret}}) \in \text{Calls}, \ a_{\text{ret}} \equiv k \odot p.\text{name} \quad [\text{obj-get}]
\]

\[
p \in L(k,*) , \ p.\text{set} = \text{obj}(k') \implies \ (k', k, a_{\text{arg}}, a_{\text{ret}}) \in \text{Calls}, \ a_{\text{arg}} \odot 0 \equiv k \odot p.\text{name} \quad [\text{obj-set}]
\]
For properties that contain primitive values, we add the primitive to the types of the corresponding term:

\[
\text{If } p \in \mathcal{L}(k, *), \ p.\text{value} = x, \ \text{IsPrimitive}(x) \quad \text{then } x \in \text{Types}(k \diamond p.\text{name}) \quad \text{[obj-primitive]}
\]

Finally, the function objects are registered in \( Funs \):

\[
k \in \mathcal{O} \quad \frac{s(k).\text{function} \in Funs(k)}{\text{[obj-func]}}
\]

### 8.5.10 Assignment Compatibility

TypeScript defines assignment compatibility between types as a more liberal version of subtyping. It can be seen as a bi-directional subtyping check, that is, two types are assignment compatible if either type is a subtype of the other. When checking object types for assignment compatibility, the rule is applied recursively for each property. As an example, the types \{opt \( x : \text{num}; y : \text{num} \} \) and \{opt \( z : \text{num}; y : \text{num} \} \) are assignment compatible despite neither being a subtype of the other.

We extend the notion of assignment compatibility to handle atoms and values (the two new types introduced in this section). The full definition is too verbose for this presentation, so we will briefly describe how atoms are checked against types. When checking an atom \( a \) (or any term that has been unified with an atom \( a \)) for assignment compatibility with an object type \( o \), we recursively check the types of \( a \diamond f \) against the type of every property \( f \) on \( o \). When checking the type of a non-atom term \( t \) against a type \( \tau \), the check succeeds if any of the types in \( \text{Types}(t) \) is assignment compatible with \( \tau \).

### 8.6 Evaluation

Our approach is implemented in the tool chain \textsc{jsnap}[^1], \textsc{tscore}[^2] and \textsc{tscheck}[^3] for producing snapshots, extracting TypeScript Declaration Core declarations from declaration files, and checking a snapshot against the declarations, respectively. All tools are implemented in JavaScript.

With these tools, we conduct experiments to establish the usefulness of our approach. Specifically, we wish to determine how many warnings our technique produces in practice, and how many of these warnings are indicators of real bugs.

We selected the 10 largest declaration files from the Definitely Typed repository (mentioned in the introduction) for which we were able to obtain the correct version of the library code. We ran our tool on each benchmark and manually classified each warning as one of the following:

- **Declaration:** This warning is due to a bug in the declaration file.
- **Library:** This warning is due to a mismatch between the library implementation and its official documentation.

[^1]: https://github.com/asgerf/jsnap
[^2]: https://github.com/asgerf/tscore
[^3]: https://github.com/asgerf/tscheck
8.6. Evaluation

<table>
<thead>
<tr>
<th>benchmark</th>
<th>lines of code</th>
<th>decl</th>
<th>lib</th>
<th>spurious</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.js .d.ts</td>
<td>H</td>
<td>L</td>
<td>I</td>
</tr>
<tr>
<td>ace</td>
<td>13,483</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fabricjs</td>
<td>8,584</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jquery</td>
<td>6,043</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>underscore</td>
<td>885</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>pixi</td>
<td>5,829</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>box2dweb</td>
<td>10,718</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>leaflet</td>
<td>5,873</td>
<td>13</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>threejs</td>
<td>19,065</td>
<td>49</td>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td>d3</td>
<td>9,225</td>
<td>21</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sugar</td>
<td>4,133</td>
<td>(37)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>142</td>
<td>39</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 8.1: Classification of warnings on the 10 benchmarks.

**Spurious:** This is a false warning, i.e. it does not indicate an actual bug in the declaration file or library code.

Note that when multiple warnings are due to the same bug in the declaration file, they are only counted as one, since duplicate warnings provide no additional value to the user.

We observe that some bugs in the declaration file are of little significance in practice, so we further classify the declaration warnings according to their impact:

**High impact:** The bug causes tools to misbehave, e.g. the TypeScript compiler may reject correct client code, or code completion may provide misleading suggestions.

**Low impact:** Although the declaration is technically incorrect, the TypeScript compiler and its code completion services will behave as intended. A typical example of this is a boolean property that should be declared optional.

Likewise, the spurious warnings are also classified further:

**Initialization:** The warning is due to the initialization assumption, i.e. parts of the API are not available until some other functions have been called or an event has fired.

**Platform:** The warning is due to the platform assumption, i.e. parts of the API are deliberately only made available on certain platforms.

**Unsound analysis:** The warning was produced because the static analysis did not infer a proper over-approximation of a function’s possible return values and the compatibility check subsequently failed.

The results are shown in Table 8.1. The $H$ and $L$ columns show the number of declaration bugs found, respectively of high impact and low impact. The $lib$ column shows the number of library bugs found. The $I$, $P$, and $U$ columns show the number of spurious warnings due to initialization, platform, and unsound analysis, respectively.

With 142 high-impact bugs, this evaluation shows that our technique is capable of finding a large number of bugs in TypeScript declaration files, and even detect some bugs the library code. Only 22% of the warnings are spurious, most of which pertain to the
8. Checking Correctness of TypeScript Interfaces for JavaScript Libraries

sugar library, which uses some native JavaScript functions in a way that our analysis currently does not support.

The running time is dominated by the static analyzer, which took an average of 1.1 seconds per checked function signature on a Intel Core i3 3.0 GHz PC. The average analysis time was 176 seconds per benchmark, with the slowest being the sugar library whose 420 function signatures took 628 seconds to check.

Of the high-impact bugs, 68 were detected by the snapshot analysis (Section 8.4), and the remaining 74 were detected by the static analysis (Section 8.5).

8.6.1 Examples

A recurring type of high-impact bug was found when the result of a function gets passed through an asynchronous callback rather than being returned immediately. For instance, in fabric.js, we found the declaration (simplified here):

```
118 declare var Image : { ...
119     fromURL(url: string): IImage;
120 };
```

Image.fromURL in fact takes a callback argument to be invoked when the image has been loaded (asynchronously); the function call itself returns nothing.

An example of a spurious warning was found in the following fragment of the leaflet library (simplified):

```
121 // declaration file:
122 declare var DomUtil : { ...
123     setPosition(el: HTMLElement, point: Point);
124     getPosition(el: HTMLElement): Point
125 };
126 // implementation:
127 var DomUtil = { ...
128     setPosition: function (el, point) {
129         el._leaflet_pos = point;
130         /* rest of code omitted */
131     },
132     getPosition: function (el) {
133         return el._leaflet_pos;
134     }
135 }
```

A comment in the source code suggests that getPosition should only be used after setPosition and based on this rationale assumes that the position stored in setPosition line 129 is available. However, since we check each function signature in isolation, our static analysis does not observe the field established in setPosition when checking getPosition. The expression el._leaflet_pos is thus determined to return undefined, which fails to check against the Point type, and our analysis thus issues a spurious warning regarding the return type of getPosition.

8.6.2 Limitations in TypeScript

The high-impact warnings for the sugar library are shown in parentheses because it is clear from the declaration file that the developer is already aware of all of them. The bugs are currently unfixable due to a limitation in TypeScript that makes it impossible to extend the type of a variable that was defined in TypeScript’s prelude declaration file (the
prelude is implicitly included from all other declaration files, similar to the `java.lang` package in Java). These warnings were all reported by the first phase of the algorithm.

We observed another limitation of TypeScript in the declaration file for the `leaflet` library. This library configures itself based on the presence of certain global variables, such as `L_DISABLE_3D`, intended to be set by the client. It is helpful for the declaration file to document these variables, but the variables are absent by default, and global variables cannot be declared optional in TypeScript (though our core language allows it). To spare the user from such low-impact warnings that cannot be fixed, we configured our analyzer to omit all warnings regarding missing boolean properties if the property occurs in a position where it cannot be declared optional.

Lastly, we observed that the `threejs` library declares several enum types with no members, because the members of the enum are not encapsulated in a namespace object like TypeScript expects, as shown in the following fragment:

```typescript
enum CullFace { }
var CullFaceBack: CullFace;
var CullFaceFront: CullFace;
```

To compensate, we configured our analyzer to treat empty enums as the `any` type.

None of these limitations exist in TypeScript Declaration Core (Section 8.2.1), so a more liberal syntax for TypeScript would alleviate the issues.

8.7 Related Work

Most high-level programming languages provide interfaces to libraries written in low-level languages. The challenge of ensuring consistency between the low-level library code and the high-level interface descriptions has been encountered for other languages than TypeScript and JavaScript. Furr and Foster have studied the related problem of type checking C code that is accessed via the foreign function interfaces of Java or OCaml [30]. Due to the major differences between JavaScript and C and between the notion of types in TypeScript compared to those in Java or OCaml, their type inference mechanism is not applicable to our setting. St-Amour and Toronto have proposed using random testing to detect errors in the base environment for Typed Racket numeric types, which are implemented in C [81]. We believe it is difficult to apply that technique to TypeScript types and JavaScript libraries, since TypeScript types are considerably more complex than the numeric types in Typed Racket. Another essential difference between these languages is that JavaScript libraries are initialized dynamically without any static declaration of types and operations, which we have chosen to address by the use of heap snapshots.

The dynamic heap type inference technique by Polishchuk et al. [65] is related to our snapshot type checking phase (Section 8.4), but is designed for C instead of TypeScript, which has a different notion of types. Unlike C, JavaScript and TypeScript are memory-safe languages where all runtime values are typed (with JavaScript types, not TypeScript types), and we can disregard the call stack, which leads to a simpler algorithm in our setting.

To the best of our knowledge, no previous work has addressed the challenge of automatically finding errors in library interface descriptions for JavaScript, nor for dynamically typed programming languages in general. TypeScript is by design closely connected to JavaScript, but other statically typed languages, such as Dart [33] or Java via GWT [32],
contain similar bindings to JavaScript, to enable applications to build upon existing JavaScript libraries. Despite the young age of TypeScript, the immense volume of the Definitely Typed repository testifies to the popularity of the language and the importance of interacting with JavaScript libraries.

Several more expressive type systems for JavaScript than the one in TypeScript have been proposed, including TeJaS [56] and DJS [15]. We only use TypeScript’s meaning of types, not its rules for type checking program code, and it may be possible to adapt our system to such alternative type systems, if they were to introduce language bindings to JavaScript libraries.

The static analysis in Section 8.5 is a unification-based analysis in the style of Steensgaard [82] and bears some resemblance to the one used in our previous work on refactoring [24], although that work did not deal with function calls, prototype-based inheritance, dynamic property access, heap snapshots, or types. Many other static analysis techniques have been developed for JavaScript, for example, TAJS, which is designed to detect type-related errors in JavaScript applications [46], WALA [80], and Gatekeeper [36]. None of these incorporate type declarations into the analysis, and they focus on soundness, not performance.

A potential alternative to our approach could be to apply hybrid type checking [53] by instrumenting the program code to perform runtime checks at the boundary between TypeScript applications and JavaScript libraries. By the use of static analysis, we avoid the need for instrumentation and high-coverage test suites.

8.8 Conclusion

We have presented an approach to automatically detect errors in TypeScript declaration files for JavaScript libraries. Our key insights are that the API of such a library can be captured by a snapshot of the runtime state after the initialization and that the declarations can be checked on the basis of this snapshot and the library function code using a light-weight static analysis. Our experimental evaluation shows the effectiveness of the approach: it successfully reveals errors in the declaration files of most of the libraries we have checked using our implementation.

In addition to being a useful tool for authors of TypeScript declaration files for JavaScript libraries, our implementation has led us to identify some mismatches between the TypeScript language specification and common uses of types in declaration files, and some practical limitations of the expressiveness of its type system, which may be helpful in the future development of the language.
Bibliography


Bibliography


[46] Simon Holm Jensen, Anders Möller, and Peter Thiemann. Type analysis for JavaScript. In International Static Analysis Symposium (SAS), 2009. [10] [48] [78] [85] [88] [92] [93] [104] [131] [160]


[48] Simon Holm Jensen, Magnus Madsen, and Anders Möller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2011. [10] [78] [85] [104] [131] [160]


[56] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Te-JaS: Retrofitting type systems for JavaScript. In Symposium on Dynamic Languages (DLS), 2013. [52] [160]

[57] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In International Conference on Compiler Construction (CC), 2003. [88] [104]


[91] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for JavaScript. In 

http://www.w3.org/DOM/DOMTR#dom3