JavaScript Type Inference Using Dynamic Analysis

Morten Passow Odgaard, 19960809

EVU Master Thesis
Master of Information Technology, Software Development, Aarhus University
June 2014
Advisor: Anders Møller

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

JavaScript has become the programming language of the web and the language is used in large systems in ways it was never designed for.

Though much progress has been made with linting tools and unit testing practices as well as static type analysis and inference, the area of generating JavaScript API annotations based on unit tests is uncharted. Tests contain type knowledge that can be made available to current code editors and analysis tools by converting the knowledge into type annotations.

This paper presents a type inference and annotation system that uses dynamic analysis based on test cases to automatically generate JSDoc type annotations for JavaScript programs and libraries. A system prototype is implemented in JavaScript and evaluated with encouraging results. The type inference method is found to be accurate in practice. Automatic generation of JSDoc type annotations shows promise although more work is required to properly annotate module-based code.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the past decade, the Javascript programming language has become pervasive in the computing industry. The main reason for this is that it is the only language that is supported in all major web browsers and can therefore be used to implement complex web applications. In addition, with the Node.js application server, Javascript can also be used on the server side and is thus a full-stack programming language.

This means that the language is now used heavily in large software systems. The language is a dynamic language and was not designed for use in large scale programs and for this reason a number of problems arise in practice. A number of measures to combat these problems, such as linting and automated tests, have gained widespread use in JavaScript tool chains in recent years. However, avoiding code errors at the time of writing requires that type information is immediately available to developers as well as to tools.

In strongly typed languages, type information is available in program source code, as per the specification of the language. The method for obtaining type information for a given source code construct is thus standardized, which allows editors to provide accurate coding assistance and compilers to perform type checking. In JavaScript this is not the case. Type information is not available in source code because the language has no constructs for expressing it. So even when types can be inferred or manually stated, a type language must be employed to convey the information to tools.

This problem is exacerbated by the fact that JavaScript has no language constructs for the concepts of modules, namespaces, classes and interfaces. These concepts are central for documentation of large code bases. In JavaScript they are implemented using a variety of patterns using functions and objects which implies an impedance mismatch between source code and the API it intends to expose.

JSDoc[20] is a mature and widely supported markup language that allows annotating JavaScript source code with type information and documentation. This enables development environments that support JSDoc to present the information to developers while they work, i.e. by providing hints and code completion. It is also partially supported by build tools such as Closure Compiler[14] to enable build warnings and errors if an annotated API is used incorrectly. Many

popular JavaScript libraries provide JSDoc annotated builds for these purposes. However, given a large un-annotated code base, generating initial JSDoc annotations is a large and manual development task.

1.2 Problem Statement

While Javascript is a dynamically typed and interpreted language, it does support programming with both types and classes. This thesis will examine the possibility of inferring static type information from JavaScript programs and libraries using dynamic analysis and test cases and present the results in the JSDoc format.

The main problem I will address is:

> Investigate whether it is feasible to use dynamic analysis in combination with test cases to automatically infer static types and generate JSDoc type annotations for program and library API's.

By defining and implementing a type inference and annotation generation system prototype, based on dynamic analysis, I will assess the practical accuracy and completeness of the technique when applied to real-world libraries.

1.3 Method

Dynamic analysis and test cases are used to infer the types used in JavaScript programs and library APIs. In order to define the type system, the notion of observed values is defined as values obtained using dynamic analysis. This notion is used to define a simple type system tailored to the requirements of API type annotations, specifically the JSDoc documentation system. An alternative type inference strategy that employs Duck Typing is defined to provide better results in some identified cases.

To obtain the values required for type inference, a specific dynamic analysis that uses a combination of static and dynamic instrumentation is specified.

To generate JSDoc type annotations, a translation that interprets and converts inferred types to relevant annotations is defined.

The system is evaluated using experimental evaluation by developing a prototype in JavaScript which is then applied to a number of libraries. The results are evaluated by measuring the accuracy of the inferred types and the generated annotations combined with an overall qualitative assessment of the system.

1.4 Overview

The rest of the paper is structured as follows:

Chapter 2 presents background information on the JavaScript language, dynamic analysis and the JSDoc documentation language.

Chapter 3 presents the type inference specification and dynamic analysis technique.

Chapter 4 discusses automatic generation of JSDoc annotations.

Chapter 5 evaluates the presented techniques. First, the prototype implementation and related tools are discussed. Secondly, a range of experiments are presented using the technique on real programs. Finally related work is discussed.

Chapter 6 concludes on the achieved results and suggests future work and improvements.

Chapter 2

Background

2.1 JavaScript

The JavaScript programming language was created in 1995 at Netscape by Brendan Eich. Because of fierce competition, a browser-based scripting solution was needed fast and a prototype was created in just 10 days which was shipped with the Netscape Navigator browser in November 1995[27].

The name JavaScript was licensed to Netscape by Sun Microsystems (now Oracle) that owned the Java trademark. The Java name remains trademarked so other implementations use other names. The language specification uses the name ECMAScript.

The language version used in this paper is ECMAScript 5.1[19].

Language

JavaScript is a dynamic language. Types are supported but not enforced, so at runtime, a variable may reference one type and later reference any other type without restriction.

Values are either primitive values or objects. The primitives are `string`, `boolean`, `number`, `null` and `undefined`. All other values are objects, including functions. Objects essentially consist of key-value pairs, called properties. Property keys are strings, numbers or identifiers and values can have any type, including functions or other objects. Properties can be changed, added and removed freely at runtime except for some built-in objects and objects that are specifically locked down.

The language has no concept of classes, namespaces or modules. Instead such concepts are commonly implemented in libraries in the language itself.

Functions are first class citizens in the language. For example, they can be assigned to variables and passed to other functions. Function overloads are not supported but can be emulated using *argument type sniffing* at runtime by examining the types of the provided arguments to determine which implementation to execute. This technique can be used to effectively provide function overloading based on both runtime types and number of arguments.

JavaScript uses prototypal inheritance and has no notion of classes. All objects are created from a prototype and have a prototype chain from which they inherit properties. This feature can be used to implement the behaviour of classical object orientation with classes and class based inheritance.

JavaScript is lexically scoped with functions as the only way of introducing additional scopes beyond the global (top level) scope. This feature is different from most (lexically scoped) languages that commonly provide block scope. However, block scope can be emulated using a function. A common pattern is the *immediately invoked function expression*, known as an IFFE[1]. This pattern is often used to provide privacy, i.e. when implementing modules.

Another important feature of functions is that of *closures*. A closure is a function combined with a connection to the scope in which it was declared. This connection provides access within the function to the free variables that were available where the function was declared. A simple example is the counter function:

Listing 2.1: Counter function utilizing a closure.

```
function createCounter(initialValue) {
  return function(increment) {
    initialValue += increment;
    return initialValue;
  };
}

var counter = createCounter(10);
console.log( counter(1) );  // outputs 11
console.log( counter(4) );  // outputs 15
```

When the inner function is returned from the `createCounter` function, the inner function keeps a connection to its free variable `initialValue` via a closure.

Thus functions are very flexible although their behaviour is also different from most mainstream languages.

Modules

Modules in JavaScript are concerned with the problem of defining and consuming encapsulated pieces of code. While the JavaScript language has no concept of modules, a number of specifications exist. The main responsibilities of module systems are:

- *Definition.* A module typically consist of a single code file. The module system specifies constraints that the code must adhere to in order to be a valid module, in particular how it should *export* its functionality.
- *Loading.* A module may be loaded (or *imported*) at runtime by consumer code. Loading the module should not affect the consumer scope, except as defined by the consumer. In particular, the consumer defines a 'namespace' for the imported code such that usage of global scope is avoided.
- *Dependencies.* A module may depend on other modules by explicitly stating its dependencies. Module systems provides a means of transitively resolving and loading dependencies.

The two most common module systems are CommonJS[1] which is supported by Node.js and Asynchronous Module Definition (AMD)[2]. AMD is derived from CommonJS and aimed at browser code by supporting asynchronous module loading. A pattern known simply as the *Module Pattern*[26] and common variations[4] hereof are also widely used.

While a thorough module system presentation is beyond the scope of this paper, a few details are necessary in later chapters. These are shown in the following example.

Listing 2.2: Example module

```
(function (exports) {

  var someLocalVariable = ...

  exports.someFunction = function(...) {...};

  exports.SomeObject = {...};

})(typeof exports === 'undefined'? this.MyModule = {} : exports);
```

The above module pattern uses a number of 'tricks' to implement a module that is supported both in browsers and in Node.js:

- Uses the special global variable `exports` that is defined by both CommonJS and AMD to export a function and an object to module consumers.
- Wraps the implementation in an IIFE to avoid polluting the global namespace.
- Defines `MyModule` as a default name in case the consumer does not specify a name when loading the module.

Modules may depend on other modules. In this case it is not generally possible to implement modules that can be used across module systems. Tools such as Browserify [3] attempt to bridge the gap by automatically translating CommonJS modules to be used in browsers.

In ECMAScript version 6, language support will be added for modules.

Host Environment

JavaScript is not a standalone language, instead it runs inside a *host environment*. The host environment is commonly a web browser but other environments are also available, such as Node.js, SpiderMonkey or Rhino. JavaScript can also be embedded inside custom applications to provide a scripting environment for controlling the application programatically.

The host environment implements a JavaScript engine, a virtual machine that interprets and executes JavaScript code and provides access to *host objects*. For example, web browsers provide access to objects that allows JavaScript code to interact with the containing web page via its Document Object Model (DOM) and network access over HTTP via the `XMLHttpRequest` object. Since code delivered via web pages cannot usually be trusted, web browsers do not provide access to low level resources such as the file system and network. The Node.js environment on

[1] `http://www.commonjs.org/specs/modules/1.0/`
[2] `https://github.com/amdjs/amdjs-api/blob/master/AMD.md`
[3] `http://browserify.org/`

the other hand can be used for general purpose programs and provides access to all resources required for those purposes, similar to scripting environments such as Perl or Ruby.

Tool Chain

In recent years much has happened in the JavaScript open source community. Driven by Node.js[4] that provides a package management system as well as general purpose operating system primitives, a multitude of tools are available that are used to combat the problems that have plagued JavaScript development - the problematic language features and the lack of a type checking compiler.

Tools are available to provide a complete build process previously only available for static languages. Examples of tools are:

- *Linting.* Linting processes source code and reports warnings and errors when the code does not conform to pre-specified constraints. Examples are violations of formatting rules and the use of undeclared variables and problematic language constructs.
- *CSS processors.* While CSS is used almost exclusively to control the layout of web pages, maintaining CSS code can be problematic for large sites. Tools such as Less and Sass provide scripting features such as variables that can limit CSS code duplication and ease maintenance.
- *Automated testing.* Automated tests are widely used, ranging from UI tests run in (headless) browsers to pure logic tests that can run in any JavaScript VM.
- *Minification and bundling.* To save bandwidth and to limit the number of HTTP requests a web page performs, the source code can be minified and bundled into single files. To be able to debug minified and/or bundled code running in production, *Source Maps* can be used.

The above makes it possible to integrate JavaScript development in a continuous integration build process.

In addition, because JavaScript code does not need compilation, tasks such as linting and automated tests can be run every time a source file is saved, providing immediate feedback. Grunt[5] is a popular tool tailored to this type of task.

A different approach to JavaScript development is that of translating or transpiling source code written in another language into JavaScript code. An early example of this is the Google Web Toolkit[6] that compiles Java code to JavaScript. This has the benefits of providing the development experience of static languages, rich code completion, type checking and automatic refactorings. The cost is increased complexity in the tool stack and build process as well as the need to work with possible semantic differences in the source and target languages. Other examples of this approach are CoffeeScript[7], Dart[8] and TypeScript (section 5.3).

[4] `http://nodejs.org/`
[5] `http://gruntjs.com/`
[6] `http://www.gwtproject.org/`
[7] `http://coffeescript.org/`
[8] `https://www.dartlang.org/`

Problematic Features

Traditionally, JavaScript has had a bad reputation, which can partly be attributed to a number of problematic features. Douglas Crockford's *JavaScript, The Good Parts*[6] discusses this at length. Some 'bad parts' are:

- *Globals.* Global variables have a number of negative implications for modularity and maintainability of large programs. It is too easy to accidentally introduce or change a global variable. While the use of globals can be minimized it cannot be avoided completely.
- *No Block Scope.* Function scope is used instead of the more common block scope. This can cause hard-to-detect bugs because a variable may belong to a different scope than it appears to, even to experienced programmers.
- *Value Coersion.* Automatic coersion of values can cause subtle bugs. The somewhat arbitrary 'truthyness' and 'falsyness' of values exacerbates this problem.
- *Automatic Semicolon Insertion.* This feature can change code semantics in unexpected ways.

Linting tools can to some degree mitigate the problems caused by these features, by issuing warnings when the features are used.

2.2 Dynamic Analysis

Dynamic analysis is the concept of executing a program in order to obtain data for analysis. Running the original program is often insufficient, so an instrumented version of the program is executed instead of the original. The instrumented program can track internal state while it runs, which is then available in a subsequent (or parallel) analysis. It is used for a variety of purposes, such as profiling, deadlock detection[22], intrusion detection and bug detection.

Dynamic analysis contrast static analysis techniques, where the source code is analysed using methods such as data and control flow analysis.

Static and dynamic analyses can be used in tandem, i.e. as in the SpiderMonkey JIT compiler used in the Mozilla Firefox web browser[16].

A common example of dynamic analysis is profiling. To obtain performance data for, say, a Java program, methods could be instrumented to write a trace message upon entrance and exit. The messages could contain information about timing and the names of the method, class and package. This would allow answering questions such as how much clock time was spend in which methods? How much CPU time? How many times were methods called and so on. This simple example illustrates that instrumentation can incur an overhead on program execution. Two main problems are often considered.

Memory

A key problem when using dynamic analysis is that to produce useful general purpose results, large amount of data needs to be generated which has a cost in terms of performance and memory consumption. For example, in embedded systems, available memory is commonly sparse which means that the generated data must be offloaded to another storage media to avoid filling up

memory and interfering with the original program behaviour. Additionally, the instrumented program itself is larger and must be able to fit into the available memory.

Performance

In systems where timing is important, such as real time systems, the instrumentation technique must be designed such that the timing guarantees of the system are not altered. Timing is important if the analysis performed is related to threads and scheduling, such as deadlocks and other race conditions.

Raw performance is also often a major consideration because the instrumentation performed may cause slowdowns in performance that are unacceptable for a production system. If the analysis in question is a bug hunt in a live system, then system must continue to provide its regular service for as long as it takes to locate the bug.

Performance is a concern in both *Pin*[23] and *Aftersight*[5].

2.3 JSDoc

JSDoc[20] is a an markup language for JavaScript source code, used to document program and library API's. Documentation can consist of both prose descriptions and structured information about the code, such as function parameter names and types. The format is accompanied by a tool that produces documentation as HTML pages. The format is similar to the JavaDoc format used for documenting Java code, but it is tailored to support (some of) the dynamic features of JavaScript.

While the original purpose of JSDoc was the generation of documentation pages, the format is now supported by many code editors to provide interactive features such as code completion and other assistance. It is also supported by non-interactive tools, notably the Closure Compiler[14] that (among other things) allows type checking and optimizations of programs based on a variation of JSDoc type annotations.

Doclets and Tags

An example of a JSDoc annotated function is shown in listing 2.3. Annotations are specified as JavaScript block comments, named *doclets*. Depending or their purpose, doclets are attached to different code entities, such as variable declarations and function declarations. The example shows a doclet attached to a function that demonstrates how type information for the function's parameter and return value is specified via syntax known as *tags* inside the doclet. A doclet can include a leading description, independent of the tags it contains.

Listing 2.3: Example of a JSDoc annotated function.

```
/**
 * Squares a number
 *
 * @param {number} x - The value to square
 * @returns {number} The square of x
 */
function square(x) {
```

```
  return x * x;
}
```

Each doclet has a specific kind that determines its purpose. The most important kinds are:

- `@member`
- `@function`
- `@class`
- `@module`
- `@namespace`
- `@typedef`

The kind tag itself is sometimes omitted because JSDoc can infer it from other tags in the doclet. For example, the doclet in listing 2.3 implicitly has the `@function` kind which is omitted because the `@param` tag is only available for functions.

Types

The supported type values are listed in table 2.1.

Type name	Example	Notes
Primitives	`string`	One of the primitive types: number, string, boolean, null, undefined
Custom type	`MyClass`	A custom type declared using ie. @class or @typedef
Built-in object type	`Date`	Class names of built-in objects
Host object type	`HTMLDivElement`	Class names of host objects
Union type	`(number|object)`	One of a possible number of types
Record type	`{a:number, b:string}`	Object having only data properties
Array	`string[]`	Typed array
Any type	`any`	
Nullable	`?number`	Either the specified type or `null`
Not nullable	`!number`	The specified type and never `null`

Table 2.1: JSDoc type overview

JSDoc also supports (to some degree) type values used by the Closure Compiler[14]. Notable extensions provided by Closure Compiler are inline function types and various forms of templated types/generics.

Namepaths

JSDoc has a *namepath* concept, used to uniquely name code entities so they can be referred from other doclets. The basic syntax is as follows:

```
someFunction
SomeClass
SomeClass.staticMember
SomeClass#instanceMember
SomeClass~innerMember
SomeNamespace.SomeOtherNamespace.someMember
module:SomeModule.SomeClass
```

By convention, classes and namespaces are in upper case, everything else is in lower (camel) case. Namepaths can be used to disambiguate references to identically named entities such as `bar` in this example:

Listing 2.4: Example code that requires using namepaths for disambiguation.

```
/** @constructor */
function Foo() {
  var bar = 'inner';      // namepath: Foo~bar
  this.bar = 'instance';  // namepath: Foo#bar
}
Foo.bar = 'static';       // namepath: Foo.bar
```

@param Tag

The `@param` tag is used to document function and method parameters. Its basic syntax is:

`@param [{<type>}] <name> [- ] [<description>]`

It can be applied multiple times for a function, once for each formal parameter. `type` is any valid type value.

There are a number of syntax variations:

`@param` {number=} x	x is an optional parameter
`@param` {number} [x=1]	x is an optional parameter with a default value of 1
`@param` {...number} x	x can be repeated any number of times

@returns Tag

The `@returns` tag is used to document a function or method return value. It is used like `@param` except that a return value is unnamed.

@type Tag

A `@type` tag is used to document an entity as having a specific type as in this example:

Listing 2.5: Example of using `@type`.

```
/** @type {number} */
var count = 0;
```

Depending on the entity, other more specific tags may be used instead, such as `@member` which is explained below.

@constructor Tag

Used on a function declaration or a variable referencing a function expression to document its purpose as a constructor. This implicitly documents a class with the name of the function or variable.

Listing 2.6: Example of a constructor function.

```
/**
 * @constructor
 * @param {number} x - The x coordinate
 * @param {number} y - The y coordinate
 */
var Point = function (x,y) {
  this.x = x;
  this.y = y;
};
```

@member Tag

Used to document a class member. Using it to document the properties of the previous example looks like this:

Listing 2.7: Example of a class using `@member`.

```
/**
 * @constructor
 * @param {number} x - The x coordinate
 * @param {number} y - The y coordinate
 */
var Point = function (x,y) {
  /**
   * Instance x coordinate
   * @member {number}
   */
  this.x = x;
  /**
   * Instance y coordinate
   * @member {number}
   */
  this.y = y;
};
```

@property Tag

An alternative to `@member` that is included in the containing class or namespace doclet instead of in its own doclet attached to the specific entity. Its syntax is identical to `@param`. An alternate version of listing 2.7 using `@property` instead of `@member` looks like this:

Listing 2.8: Example of a class using `@property`.

```
/**
 * @constructor
 * @param {number} x - The x coordinate
```

```
 * @param {number} y - The y coordinate
 * @property {number} x - Instance x coordinate
 * @property {number} y - Instance y coordinate
 */
var Point = function (x,y) {
  this.x = x;
  this.y = y;
};
```

@namespace Tag

The `@namespace` tag is applied to an object to document that the object provides a namespace for its members. It can be used to document a common pattern of using objects as a namespace mechanism. It is used when more specific tags such as `@constructor` (to document a class) or `@module` are not appropriate. It is required in order for nested properties to be documented.

Listing 2.9: Example of a namespace pattern using `@namespace`.

```
/** @namespace */
var Util = {
  /** ... */
  capitalize: function (s) {...}
};
```

The example above can also be written like this to match differently factored code:

Listing 2.10: Alternate version of listing 2.9.

```
/** @namespace */
var Util = {};

/** ... */
Util.capitalize: function (s) {...}
```

@typedef Tag

The `@typedef` tag is used to define a named type that can be referred in other doclets, ie. in `@param` or `@type` tags. The syntax is:

```
@typedef [{<type>}] <namepath>
```

It is required for complex types that are not allowed in `@type` tags, ie. objects with methods or functions with named parameters. The following example shows how a parameter with a complex type can be documented. In this case interface semantics can be expressed even though interfaces are not supported by JavaScript and not present in the source.

Listing 2.11: Example of using `@typedef` to specify a complex type.

```
/**
```

```
 * Defines an object that can be sorted by key
 * @typedef {Object} Sortable - Defines a sorting interface
 * @property {string} sortKey - The key used for sorting
 */

/**
 * @param {Sortable[]} items - The array to sort
 */
function sort(items) { ... }
```

@callback Tag

The `@callback` tag is used to document function parameters, ie. callbacks. It is equivalent to `@typedef {function}`. Its advantage over the function type syntax is that documentation can be attached to each parameter.

@enum Tag

`@enum` is used to document a number of static properties, typically constants and of the same type, as values that belong together. Can be used with `@readonly` to indicate that the values are constants.

Listing 2.12: Example of `@enum` constants.

```
/**
 * AST node names
 * @readonly
 * @enum {string}
 */
var Syntax = {
  Program = 'Program',
  VariableDeclaration = 'VariableDeclaration',
  FunctionDeclaration = 'FunctionDeclaration',
  ...
};
```

@module Tag

The `@module` tag is used to document that the code file as being a module (section 2.1). Both CommonJS and AMD modules are supported. A CommonJS example is shown below:

Listing 2.13: Example of module annotation.

```
/**
 * @module BigNum
 */

/**
 * @constructor
 * @param {(number|string)} num
 */
exports.BigInteger = function(num) {
```

```
    ...
}
```

Virtual Tags

Some tags can be used as *virtual* tags, to document entities that are not available in the source code. An example is `@typedef` as shown above that defines a complex parameter type. `@member` can also be used as a virtual tag to document class members that cannot be documented by standard means, i.e. because the member is defined dynamically by a library function.

There are other tags that change how JSDoc interpret the source code, such as `@alias`, `@name`, `@memberof` and `@instance`.

Access Modifiers

JavaScript has no language support for access modifiers. The tags `@access`, `@public`, `@protected`, `@private` and `@readonly` can be used to document the intent of access modifiers.

Additional Tags

Tags not discussed above include `@author`, `@version`, `@file`, `@deprecated`, `@copyright`, `@since` and `@see` which are all used for general documentation purposes.

Alternatives

The JSDoc documentation generation tool works by parsing annotated JavaScript source code. The parse tree and annotations are processed and the documentation pages are built.

A number of tools use a similar approach and syntax such as YUIDoc[9], Dojo DocTools[10], and ScriptDoc[11].

Two key properties of this approach are:

- *Annotate Original Source.* The annotations are provided in the source code and the available tags mainly document source code entities.
- *Oriented Towards Human Readable Documentation.* The main focus is to provide human readable documentation. There is no particular focus on type information that can be consumed by tools.

A different approach is to keep documentation separate from the original source. This approach is taken by TypeScript (section 5.3) which provides its own documentation model, named Type Definition files. This information is meant for tool usage, notably editor assistance such as code hints and refactorings (i.e. Rename symbol, Go to symbol definition). TypeScript additionally provides type checking.

[9] `http://yui.github.io/yuidoc/`
[10] `http://dojotoolkit.org/reference-guide/1.8/util/doctools/generate.html`
[11] `https://wiki.appcelerator.org/display/tis/ScriptDoc+(SDOC)+2.0+Specification`

An advantage of separating the documentation model from the source code is that the model can be self-contained and does not need to correspond to source code entities. However, a separate documentation model may also be harder to maintain.

Chapter 3

Type Inference

3.1 Specification

This chapter specifies a simple type inference mechanism by stating a number of definitions. Each definition assumes a set of input data and inductively defines how a values of a particular kind should be inferred in terms of a textual representation.

Input data is referred to as *observed data* as it is assumed to be made available via dynamic analysis. When referring to observed data, it can mean both values directly observed and types previously inferred.

The aim of the type system is to infer and represent information required to annotate programs with *static* type information. While dynamic analysis can easily capture properties being added and removed, for example, such dynamic information is not useful for annotating an API. Instead the inference specification aims to mask irrelevant dynamic behaviour and provide static type information, possibly sacrificing accuracy. This focus allows a much simpler type system than what is required for general purpose type analysis, i.e. to provide program verification. For this reason many semantic subtleties are overlooked, for example distinction between functions returning `undefined` vs. having no return statements and missing object properties vs. properties with the value `undefined`.

Type Overview

The *primitive types* in JavaScript are `number`, `boolean`, `string`, `undefined` and `null`. Values are either primitive values (having a primitive type) or objects.

Table 3.1 shows a further categorisation of the types. In particular, object values are divided into `simple`, `function`, `array` and `object` *kinds*. For completeness, the table also lists the value of the `Object.prototype.toString.call()` function used later.

The category *simple* refers to built-in JavaScript objects such as `Date` as well as host objects, i.e. `HTMLDocument` in a browser host environment. This distinction from other objects is useful because these objects can be inferred in a simple way, as we shall see shortly.

Category	Example value	Kind	Object.prototype.toString.call
primitive	`5`	`number`	[object Number]
primitive	`true`	`boolean`	[object Boolean]
primitive	`'foo'`	`string`	[object String]
primitive	`undefined`	`undefined`	[object Undefined]
primitive	`null`	`null`	[object Null]
simple	`new Date`	`Date`	[object Date]
simple	`document`	`HTMLDocument`	[object HTMLDocument]
function	`function(){}`	`function`	[object Function]
array	`[1,2,3]`	`array`	[object Array]
object	`{a:false}`	`object`	[object Object]

Table 3.1: Type overview

Type	Example type expression	Kind	Notes
class	`MyClass`	n/a	Objects created with `new`
void	`void`	n/a	No type
union	`(boolean|number)`	n/a	Either a `boolean` or a `number`

Table 3.2: Additional internal types

Table 3.2 shows additional internal types that are relevant in the inference type system. These types have no JavaScript counterpart. The `class` type is used for objects that are interpreted as class instances. `void` represents no type and is used to represent cases where no specific type can be inferred. The `union` type represents that an expression can have one of a number of different types.

Note that there is no notion of *nullable* types or optional values. Instead, this notion is modelled using union types. Nullability is problematic in JavaScript because both the types `null` and `undefined` can represent a missing value. While `null` is the obvious choice for representing an object without a value, there are cases where `undefined` is more natural. One case is that of functions, where a formal parameter for which no value is provided defaults to `undefined`. This means specifying an optional value would also need to specify whether to use `null` or `undefined` as a missing value, since a concrete implementation may not support both.

Inferred Types

The types to be inferred are the types listed in tables 3.1 and 3.2.

Their syntax is represented by the following grammar:

⟨*type*⟩ ::= ⟨*primitive-type*⟩ | ⟨*simple-type*⟩ | ⟨*union-type*⟩ | ⟨*function*⟩ | ⟨*object*⟩
| ⟨*constructor*⟩ | ⟨*class*⟩ | '`void`'

⟨*primitive-type*⟩ ::= '`boolean`' | '`number`' | '`string`' | '`null`' | '`undefined`'

⟨*simple-type*⟩ ::= '`Date`' | '`RegExp`' | ... | '`Window`' | '`HTMLDocument`'

⟨*union-type*⟩ ::= ⟨*type*⟩ | '`(`' ⟨*union-list*⟩ '`)`'

⟨*union-list*⟩ ::= ⟨*type*⟩ '`|`' ⟨*union-list*⟩ | ⟨*type*⟩

⟨*function*⟩ ::= '`fn(`' ⟨*param-list*⟩ '`)->`' ⟨*type*⟩
| '`fn(`' ⟨*param-list*⟩ '`,`' ⟨*anon-param-list*⟩ '`)->`' ⟨*type*⟩
| '`fn(`' ⟨*anon-param-list*⟩ '`)->`' ⟨*type*⟩

⟨*param-list*⟩ ::= ⟨*param*⟩ '`,`' ⟨*param-list*⟩ | ⟨*param*⟩

⟨*anon-param-list*⟩ ::= ⟨*anon-param*⟩ '`,`' ⟨*anon-param-list*⟩ | ⟨*anon-param*⟩

⟨*param*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*type*⟩

⟨*anon-param*⟩ ::= '`undefined`' '`:`' ⟨*type*⟩

⟨*array*⟩ ::= ⟨*type*⟩ '`[]`' | '`Array`'

⟨*object*⟩ ::= '`{`' ⟨*property-list*⟩ '`}`'

⟨*property-list*⟩ ::= ⟨*property*⟩ '`,`' ⟨*property-list*⟩ | ⟨*property*⟩

⟨*property*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*type*⟩

⟨*constructor*⟩ ::= '`+`' ⟨*function*⟩

⟨*class*⟩ ::= ⟨*identifier*⟩

⟨*identifier*⟩ ::= any valid JavaScript identifier

The `simple` type is used to represent built-in JavaScript objects as well as objects provided by the host environment. For a given host environment, the simple types available are fixed.

Types are considered equal if their string representations are equal.

We are now in a position to define type inference using observed values. First we need a helper operator:

Definition: *kind* operator

The *kind* operator is defined such that for any JavaScript value it returns the corresponding entry in the Kind column in table 3.1.

For example, *kind* `true` returns `boolean`.

Primitives

The type of a primitive value is readily available using the *kind* function. Thus we can define:

Definition: Primitive Type

Let v be a value with a primitive type. Then we define $T(v)$, the inferred type of v:

$T(v) := kind\ v$

We say that a value is *inferable* if its inferred type is defined.

Simple Objects

We define simple objects to be objects for which *kind* is not 'Array', 'Function' or 'Object', e.g. `HTMLDocument` and `Window`. These objects can be treated just like primitives because their names contain all the information we're interested in. We can therefore define:

Definition: Simple Type

Let v be a value with a simple type. Then we define the inferred type of v:

$T(v) := kind\ v$

It proves useful to define the *union kind* of a set of values. This corresponds closely to the union type used by JSDoc. Note that this makes sense for values of any inferable type, not only primitives and simple types.

Definition: Union Kind

Let S be a set of values that have inferable types.

The *union kind* of S is defined as:

$U(S) := \bigcup_{v \in S} T(v)$

Note that the union kind is not defined as an inferable type. It is just a (possibly empty) set of types.

We can now define the inferred union type of observed values of inferable types:

Definition: Union Type

Let e be an expression with observed values $v_1, v_2, ..., v_n$, where each v_i is inferable and let $u_e = U(\{v_j\})$.

Then inferred type of e and $\{v_j\}$ is defined as:

$$T(e) = T(\{v_j\}) := \begin{cases} (t_1|t_2|\ldots|t_m)\ , t_i \in u_e & \text{if } u_e \neq \emptyset \\ \texttt{void} & \text{otherwise} \end{cases}$$

For example, for observed values `3`, `4` and `false`, the inferred type is the string `(number|boolean)`.

`void` acts as neutral element for type union. This corresponds with the intuition that if no types are observed, the `void` type is inferred.

Functions

Assume that we know how to infer the types of parameters and return values observed for a function. An intuitive way of inferring the function's signature is then to infer each parameter type as the union type of its observed values. Likewise the return type can be inferred as the unit type of the observed return values. It gets slightly more complicated when formal parameters are taken into account. Formal parameters for which no values are observed are inferred as `undefined` in correspondence with JavaScript semantics. Observed parameter values for which no formal parameter exist are added to the parameter list using an `undefined` name.

Definition: Function Type

Let f be a function declaration or function expression with formal parameter names $p_1, \ldots, p_m$ (that is not observed to be invoked using `new`).

Denote observed parameter values at index i by $\{v_j\}_i$ and observed return values by $\{r_j\}$ and let M denote the largest parameter index observed.

Now let t_i be the inferred union type for observed values $\{v_j\}_i$ and t_{return} the inferred union type for observed values $\{r_j\}$.

Then the inferred type of f is defined as:

If $m < M$:
$T(f) := fn(p_1 : t_1, \ldots, p_m : t_m, undefined : t_{m+1}, \ldots, undefined : t_M) \rightarrow t_{return}$

If $m = M$:
$T(f) := fn(p_1 : t_1, \ldots, p_m : t_m) \rightarrow t_{return}$

If $m > M$:
$T(f) := fn(p_1 : t_1, \ldots, p_M : t_M, p_{M+1} : undefined, \ldots, p_m : undefined) \rightarrow t_{return}$

For example, the function `function f(a){}` invoked with `f(1, false)` is inferred as `fn(a:number, undefined:boolean) -> void`.

Note that the union of function types will collapse only if the functions have identical types, i.e. no attempt is made to merge functions with different signatures.

In the section Duck Typing, an alternate function type definition is provided.

Arrays

If the values in an array are inferable, the type of the array can be inferred as an array of values of the union type. This is not considered very useful, however, if the union type contains more than one type. In that case a simple 'untyped' array is inferred. This is summarized below.

Definition: Array Type

Let a be an array containing inferable values $\{v_j\}$.

Let $M = \mid U(\{v_j\}) \mid$ be the number of inferred types of the values of a and let t be the inferred union type of $\{v_j\}$.

Then the inferred type of a is defined as:

$$T(a) = \begin{cases} t[] & \text{if } M = 1 \\ Array & \text{otherwise} \end{cases}$$

Objects

Assume that we can infer the types of properties observed for an object (not a simple object, array or function). The object's type is then defined as follows.

Definition: Object Type

Let o be an object with observed property keys $\{p_1, \ldots, p_n\}$ and let $\{v_j\}_{p_i}$ be the values observed for the property with key p_i.

Let t_{p_i} be the inferred union type for observed values $\{v_j\}_{p_i}$.

The inferred type of o is then defined as:

$$T(o) := \{p_1 : t_{p_1}, \ldots, p_n : t_{p_n}\}$$

Note that this definition is unable to capture changes to the type caused by properties being dynamically added or removed. As the definition does not take into account *when* properties are observed, a property is included in the inferred type regardless of when it was added to the object. If a property is removed, this is also not reflected in the inferred type. In general this means that the inferred type may not accurately represent the object at all times, or even at any time. However, considering the aim of our analysis and that the removal of properties is not common behaviour of objects that are exported as part of an API, it is hypothesised that the behaviour defined above is accurate in practice. Even if an object is defined incrementally by adding properties, the object is assumed to be exported (see section 2.1) in its 'completed shape', in which case the inferred type will be accurate.

As for functions, the union of object types will collapse only if the objects are equal, that is, if they contain the same property names and types.

The section Duck Typing below specifies an alternate function type definition which also includes a modification to how objects are inferred when used as function parameters.

Classes

Classes are not directly supported in the JavaScript language. Instead class-like behaviour can be implemented using various patterns. In the following we consider only classes defined by a constructor function that is observed to be invoked using the `new` keyword:

Listing 3.1: Simple class pattern.

```
function Point(x,y){
  this.x = x;
  this.y = y;
}

var p = new Point(2,3);
```

Other means of instantiating classes, e.g. by using library functions are not considered.

> Definition: Constructor Function Type
>
> Let f be a function declaration or function expression observed to be invoked as a constructor using the `new` keyword.
>
> Then the inferred type of f is defined as:
>
> $T(f) := +T'(f)$, where $T'(f)$ refers to the inferred function type of f.

The inferred type of an object created by a constructor function is just the class name:

> Definition: Class Type
>
> Let o be an object observed to be created by a constructor function with name C.
>
> Then the inferred type of o is defined as:
>
> $T(o) := C$

The definitions above allows inferring constructor functions and classes but does not provide useful information about the shape of class instances, i.e. properties and their types. This information is available as the object type of class instances, but there a various options for defining exactly which properties the *class* contains in case instances do not have the same properties. The definition below states that only properties present on all observed instances are included. This makes sense because it is much more common to add properties than it is to delete them. It also captures the concept of a class being relatively static compared to standalone objects.

Definition: Class Properties

Let $o_1, \ldots, o_n$ be objects with inferred class type C. Let $p_1, \ldots, p_m$ be the intersection of the property keys observed for the objects and let $\{v_j\}_{p_i}$ be the observed values for p_i for all of $o_1, \ldots, o_n$.

Now let t_{p_i} be the inferred union type for observed values $\{v_j\}_{p_i}$.

Then the class C has the properties $p_1, \ldots, p_m$ where p_i has type t_{p_i}.

At this point all JavaScript values are inferable (by induction), provided that they are constructed from inferable types. This leaves out only self-referring construction chains, such as an object having a property that refers the object itself.

For now such values remain 'un-inferable' but we shall see that this is not a problem in practice.

Duck Typing

Duck Typing is a loosely defined concept named from the saying "if it walks like a duck and quacks like a duck, then it probably is a duck", meaning that as long as the required properties of an object are available and of appropriate types, then the object is valid for a given operation. It is used heavily in JavaScript, presumably because it enables structured objects to be easily created and passed around without having to define the structure as a class. It is simple and "low ceremony" but also potentially error prone because a simple typing/spelling error can go undetected and result in errors much later in the program flow. The concept contrasts that of classes in strongly typed languages, where a compiler can guarantee that objects have a certain structure.

Consider the following example where a function is passed as an argument to another function. This is a common scenario for callback functions in asynchronous APIs.

Listing 3.2: Function invoked in different scopes.

```
function foo(a) {}
function bar(f, x) { f(x) }
foo(false);
bar(foo, 2);
```

Since `foo` is invoked with both a boolean parameter and a number parameter, its type will (using the definition above) be inferred as `fn((boolean|number)) -> void`. This is considered the *global type of `foo`*.

However, in the context of `bar`, its `f` parameter is only invoked using a number parameter so in the spirit of Duck Typing, `f`'s type should instead be `fn(number) -> void`. This is considered the *type of `foo` in the context of `bar`*.

This inference behaviour can be realized by restricting the set of observed parameter values of `foo` such that only observations that take place *while `bar` is executing* are used in the inference of `f`. This effectively infers different types of `foo` and `f`. Note that Duck Typing only applies

to function parameter and not return values. In case a function returns another function, the returned function's type should be global and only invocations of it should be subject to Duck Typing.

The above also applies to objects when used as function parameter (except classes whose types are considered static). In this case, we're only interested in observing property access during function execution. This defines a local type of the object, representing only functionality actually used by the function.

Defining the above behaviour is a simple modification to the previous definition of inferred function types:

Definition: Function Duck Type

Let f be a function declaration or function expression with formal parameter names $p_1, \ldots, p_m$.

Denote parameter values *observed during execution of* f at index i by $\{v_j\}_i$ and observed return values by $\{r_j\}$ and let M denote the largest parameter index observed.

Now let t_i be the inferred union type of $\{v_j\}_i$ and t_{return} be the inferred union type of $\{r_j\}$.

Then the inferred duck type of f is defined as:

If $m < M$:
$T(f) := fn(p_1 : t_1, \ldots, p_m : t_m, undefined : t_{m+1}, \ldots, undefined : t_M) \rightarrow t_{return}$

If $m = M$:
$T(f) := fn(p_1 : t_1, \ldots, p_m : t_m) \rightarrow t_{return}$

If $m > M$:
$T(f) := fn(p_1 : t_1, \ldots, p_M : t_M, p_{M+1} : undefined, \ldots, p_m : undefined) \rightarrow t_{return}$

Note that *observed during execution of* f also implies that object and function parameters are (recursively) inferred using the restricted set of observed values. I will refer to the usage of the Function Duck Type definition as *duck-mode*.

3.2 Dynamic Analysis

Dynamic analysis will be used to obtain the *observed data* used by the definitions in the previous section. This section describes a dynamic analysis technique that provides the necessary data. The technique consists of these steps:

- *Instrumentation.* A combination of source code instrumentation and dynamic proxies are used to produce a trace log when the program is executed.
- *Execution.* The instrumented code is exercised by test code. This populates the trace log with information about the actual values used by the program during execution. The test code can be written specifically for the purpose of inferring types or it can be a test suite that already exist for the program.

Figure 3.1 shows an overview of the source code instrumentation process. While processing the abstract syntax tree (AST) of the source code, a unique identifier is assigned to each global variable declaration and to each function declaration, function expression and object instantiation. A scope map object is generated that maps the identifiers assigned to information about the AST nodes in the original source. This data is stored for use during type inference. The information includes whether the corresponding symbol is global (a global variable or function) and its name (variable or function name).

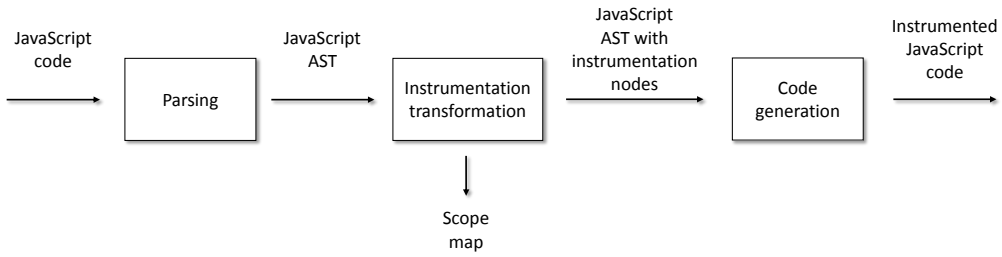

Figure 3.1: Instrumentation phase.

Figure 3.2 shows the execution and type inference flow that infers the program's types.

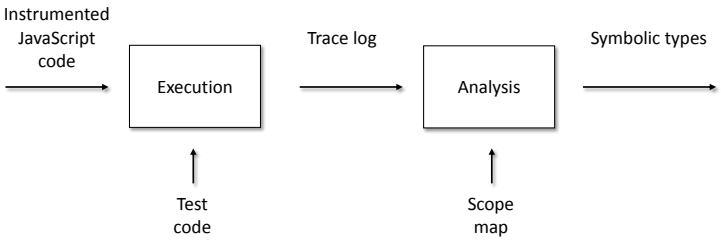

Figure 3.2: Execution and analysis (type inference) phase.

Instrumentation

In order to produce an analysable trace, the program source code is instrumented by inserting function calls that write tracing information to a log when the program is executed. This is done by parsing the original source code into an AST, transforming the AST to include trace call nodes and finally generating the instrumented source code.

The Jalangi[28] framework uses a very comprehensive instrumentation which is required for the purpose of accurate shadow execution and replay.

While this thesis uses a similar technique, a much simpler instrumentation is used because much less tracing information is necessary for type inference than for general purpose analysis. In particular, since the purpose is to generate API documentation, the types of local variables are not relevant.

Table 3.3 lists the different kinds of trace log entries that the instrumentation should facilitate.

Entry	Content
`assign`	Variable id and value assigned
`enter`	Function id, parameter values, function value
`returning`	Function id, return value
`create`	Object id, class name if applicable
`ctor`	Object id, constructor value
`prop`	Object id, property name and value

Table 3.3: Trace log entries and content

Variables

The instrumentation necessary to produce `assign` entries is shown in listing 3.3. Only global variables are instrumented.

Listing 3.3: A variable declaration and its instrumented version

```
var a = false;

var a = Trace.assign(1, false);
```

Note that to keep the examples simple, *the instrumentation code for variables is omitted for the remainder of this chapter.*

Functions

The instrumentation of functions that is required to produce the necessary data (`enter` and `returning`) is outlined in listing 3.4 and 3.5.

Listing 3.4: A function declaration and its instrumented version

```
function add(a, b) {
```

```
  return a + b;
}

function add(a, b) {
  Trace.enter(1, arguments, arguments.callee);
  return Trace.returns(1, a + b);
  Trace.returns(1, undefined);
}
```

Listing 3.5: A function expression and its instrumented version

```
var add = function (a, b) {
  return a + b;
}

var add = function (a, b) {
  Trace.enter(1, arguments, arguments.callee);
  return Trace.returns(1, a + b);
  Trace.returns(1, undefined);
}
```

`Trace.enter` keeps track of the fact that the function was called, the assigned function identifier, its parameters and its definition. This statement is inserted as the first statement in the function body. The `Trace.returns` call logs the return value before returning it and is inserted for each return statement. As the final body statement, another `Trace.returns` call is inserted to keep track of scope for functions that do not return a value.

Note that except for `return` statements, the original function body is not instrumented.

The original behaviour of the function is not affected by the instrumentation, except in the case where it throws an exception. This case is not essential and is not handled because it requires significantly more complex instrumentation.

Objects

For object types instrumentation is performed as a combination of source code instrumentation and dynamic proxies. The source code instrumentation to produce a `create` entry is very simple:

Listing 3.6: Object instantiation and its instrumented version

```
var obj = { a: false };

var obj = Trace.create(1, { a: false }, null);
```

In addition to instantiation via object literals as shown above, instantiations using `new Object()` and `Object.create()` are also instrumented.

When the `Trace.create` function is invoked, the object provided as the second argument is wrapped in a proxy object which is then returned. The proxy forwards property access to the wrapped object so that external object behaviour is unchanged and writes a `prop` entry to the trace log upon property access.

The last parameter (`null`) is used to track constructor functions as shown in the following example. This allows tracking class instances. When a class name is provided to `Trace.create`, an additional `ctor` entry is generated that allows associating the object id with the class name and the constructor function.

Listing 3.7: Instantiation of a class and its instrumented version

```
function Point(x,y) {
  this.x = x;
  this.y = y;
}

var p = new Point(0,0);

var p = Trace.create(1, new Point(0, 0), 'Point');
```

Execution

The instrumented code is executed by running test cases for the program. Test code should also be instrumented such that objects or functions created by the tests are traced and their types inferred when used in the program. When the instrumented code runs, the tracing statements are executed and entries are written to the trace log.

Listing 3.8 shows the instrumented code from listing 3.4 with added test code that invokes the function. The resulting trace log data is shown in the JSON format in listing 3.9. The value `"func": "<function reference>"` means that a reference to the actual function instance is added to the trace log. This allows subsequent analysis to obtain the id assigned to a function from the function instance. This also implies that the trace log is only analysable from JavaScript code and it cannot be serialized.

Listing 3.8: Instrumented function and test code

```
function add(a, b) {
  Trace.enter(1, arguments, arguments.callee);
  return Trace.returns(1, a + b);
  Trace.returns(1, undefined);
}

// test code
add(2, 3);
```

Listing 3.9: Trace log for listing 3.8

```
[
  { "op": "enter", "id": 1, "func": "<function reference>", "args": [2, 3] },
  { "op": "returning", "id": 1, "returnValue": 5 }
]
```

Tracing of object access is handled by the proxy object that each object created is wrapped in. The proxy provides tracing information for each property read and write.

Listing 3.10 shows the instrumented code from listing 3.6 with added test code that accesses a property. The resulting trace log data is shown in listing 3.11. Properties defined in the object literal are also added to the trace log by the `Trace.create` method.

Listing 3.10: Instrumented Object instantiation and test code

```
var obj = Trace.create(1, { a: false }, null);

// test code
obj.b = 2;
```

Listing 3.11: Trace log for listing 3.10

```
[
  { "op": "create", "id": 1, "className": null },
  { "op": "prop", "id": 1, "prop": "a", "value": false },
  { "op": "prop", "id": 1, "prop": "b", "value": 2 }
]
```

When a constructor function is invoked, an additional trace method is involved to establish the constructor-class-instance relationship. Executing the code in listing 3.7 results in the following trace log:

Listing 3.12: Trace log for listing 3.7

```
[
  { "op": "enter", "id": 1, "func": "<function reference>", "args": [0, 0] },
  { "op": "returning", "id": 1 },
  { "op": "create", "id": 2, "className": "Point" },
  { "op": "ctor", "func": "<function reference>", "objectId": 2 },
  { "op": "prop", "id": 2, "prop": "x", "value": 0 },
  { "op": "prop", "id": 2, "prop": "y", "value": 0 }
]
```

3.3 Discussion

The type system specified in section 3.1 can be characterized as structural as it is based on recursive union types. Type inference is performed by selectively collecting observed type data from the trace log to build the type structures. In comparison, the Rubydust[2] system (discussed in section 5.3) generates constraints from observed behaviour, and constraint solving is used to infer types. Their technique infers more general types than simply a type of a variable, say a class A. Instead they might infer the more general 'has a property `a` of type `number`' which can be thought of as an interface type in Java/C# or a `@typedef` in JSDoc. For this particular case, these same two notions are captured by the definitions in 3.1, in non-duck-mode and duck-mode respectively.

The authors of Rubydust prove a theorem stating that if all paths of a method are traversed during inference, then the inferred type is correct. This *all-paths-traversed* criterion is clearly also necessary for the system presented here as each branch may include a `return` statement returning values of different types and to capture these types, all branches must be traversed. They also note that when using test-driven development, tests should ensure that all branches are covered. In practice, however, this is often not necessary as even a single test case may produce an accurate inference result. It is hypothesized that at API boundaries, this is a common case.

The all-paths-traversed criterion is not sufficient for accurate inference in the presented system. A simple counterexample where the system fails to produce accurate results regardless of test inputs is the following:

Listing 3.13: Uninferable function example

```
function getProperty(o, p) {
  return o[p];
}
```

As the return value can have any type which depends entirely on the input values, its signature cannot be accurately inferred in the general case using *observed values*.

Note however, that while the function is both useless and not accurately inferable in the general case, in cases where it caries additional semantics, it might both useful and accurately inferable. Consider the following example that is technically equivalent, but carries domain semantics of a web site shopping domain. The signature shown can be inferred accurately using two test cases that would certainly be present in a system with decent test coverage.

Listing 3.14: Semantic function example

```
/**
 * @param {(Product|Item)} entity
 * @param {string} kind
 * @returns {number}
 */
function getPrice(entity, kind) {
  return entity[kind];
}
```

Another case where the all-paths-traversed criterion is insufficient, is that of generic methods. A generic method might accept a value of any type and return a value of the same type. This case is also not supported well by the presented system. Regardless of how many tests using different types are provided, the inferred input and output types will just be union types with no correlation of input and output types. And even if the system supported correlation of parameter and return value types, using that as an inference technique would be merely a heuristic. In contrast, most static analysis techniques handle these cases well by understanding the data flow of the method's implementation.

Chapter 4

JSDoc Generation

4.1 Introduction

This chapter investigates how type information inferred using the technique specified in the previous chapter can be used to automatically generate JSDoc type annotations.

Figure 4.1 shows the data flow of producing JSDoc output. Since JSDoc annotations are merged with source code and depend on code syntax, both inferred types and an AST are given as input to the transformation.

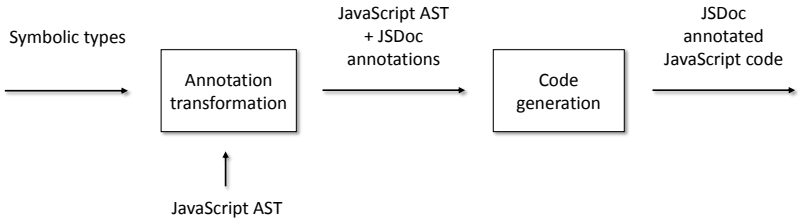

Figure 4.1: JSDoc generation

Languages such as Java and C# have a natural API description hierarchy from the top level of packages/namespaces to interfaces and classes to the lowest level of methods and fields. These levels are supported by constructs in the languages.

Similarly, JSDoc includes annotations for modules, namespaces, classes, methods, functions and variables.

The hierarchy of members in a JavaScript API is more difficult to identify than in most statically typed languages, because at the top level, the language contains only variables, functions and objects. We have seen that also classes (with some restrictions) can be identified but namespaces and modules can only be implemented using patterns of functions and/or objects so they cannot be easily identified - either from source code/AST or from inferred types.

4.2 Specification

This section specifies how JSDoc annotations are generated. API constructs included are global variables (of all types), global functions and classes (as defined in section 3.1). Namespaces and modules are not included.

The following grammar defines the syntax of the annotation language. Some symbols are carried over from the type inference grammar in section 3.1. Symbols prefixed with *jsd* indicate JSDoc type expressions and the suffixes *doclet* and *tag* indicates JSDoc doclets and tags respectively. The grammar does not reflect that annotations are intermingled with source code.

⟨*jsdoc-api*⟩ ::= (⟨*type-doclet*⟩ | ⟨*function-doclet*⟩ | ⟨*typedef-doclet*⟩) *

⟨*type-doclet*⟩ ::= '`/** @type {`' ⟨*jsd-type*⟩ '`}`' '`*/`'
| '`/** @type {object}`' ⟨*property-tag*⟩* '`*/`'

⟨*jsd-type*⟩ ::= ⟨*primitive-type*⟩ | ⟨*simle-type*⟩ | ⟨*jsd-reference*⟩ | ⟨*jsd-union*⟩

⟨*jsd-reference*⟩ ::= ⟨*object-ref*⟩ | ⟨*function-ref*⟩ | ⟨*class-ref*⟩

⟨*jsd-union*⟩ ::= '`(`' ⟨*jsd-union-list*⟩ '`)`'

⟨*jsd-union-list*⟩ ::= ⟨*jsd-union-elm*⟩ | ⟨*jsd-union-elm*⟩ '`|`' ⟨*jsd-union-list*⟩

⟨*jsd-union-elm*⟩ ::= ⟨*primitive-type*⟩ | ⟨*simple-type*⟩ | ⟨*jsd-reference*⟩

⟨*property-tag*⟩ ::= '`@property {`' ⟨*jsd-type*⟩ '`}`' ⟨*identifier*⟩

⟨*function-doclet*⟩ ::= '`/**`' ⟨*function-signature*⟩ '`*/`'
| '`/** @constructor`' ⟨*function-signature*⟩ '`*/`'

⟨*function-signature*⟩ ::= ⟨*param-tag*⟩* ⟨*opt-returns-tag*⟩

⟨*param-tag*⟩ ::= '`@param {`' ⟨*jsd-type*⟩ '`}`' ⟨*identifier*⟩

⟨*returns-tag*⟩ ::= '`@returns {`' ⟨*jsd-type*⟩ '`}`'

⟨*typedef-doclet*⟩ ::= ⟨*typedef-object-doclet*⟩ | ⟨*typedef-function-doclet*⟩

⟨*typedef-object-doclet*⟩ ::= '`/** @typedef {object}`' ⟨*identifier*⟩ ⟨*property-tag*⟩* '`*/`'

⟨*typedef-function-doclet*⟩ ::= '`/** @typedef {function}`' ⟨*identifier*⟩
⟨*function-signature*⟩ '`*/`'
| '`/** @typedef {function}`' ⟨*identifier*⟩
@constructor ⟨*function-signature*⟩ '`*/`'

⟨*member-doclet*⟩ ::= '`/** @member {`' ⟨*jsd-type*⟩ '`}`' '`*/`'

As noted above, both inferred types and the program AST is provided as input to the generation of annotations and the context of the specification includes both these elements. Inferred types are used for type information including the top level types and their names. The AST, enriched with location information is required because some annotations must be applied to specific symbols, for example ⟨*member*⟩ doclets. The AST allows discovering the correct nodes and the location information provides the matching location (line,column) in the source code, such that annotations can be inserted at that location.

At the syntactical level, two node types in the program body are annotated, *VariableDeclarator* nodes and *FunctionDeclaration* nodes, corresponding to global variables and functions. These annotations are represented by ⟨*type-doclet*⟩ and ⟨*function-doclet*⟩. The third top level annotation, ⟨*typedef-doclet*⟩ represents a virtual doclet that is not related to a source code element. Global variables or functions introduced in other ways are not annotated directly but possibly indirectly by a ⟨*typedef-doclet*⟩ as described below.

Annotation of Variables

Global variables are annotated with either a ⟨*type-doclet*⟩ or a ⟨*function-doclet*⟩.

For primitive types, simple types and union types, the `@type` doclet is used with the type expression listed in tables 3.1 and 3.2 while `void` types are not annotated. Note that ⟨*jsd-union*⟩ types is a restriction of the inferred union type ⟨*union-type*⟩ that cannot contain function or object types. In case a ⟨*union-type*⟩ contains functions or objects, these are replaced with ⟨*jsd-reference*⟩ symbols in order to produce a ⟨*jsd-union*⟩. ⟨*jsd-reference*⟩ are JSDoc namepaths which are named references to exported types (see Type Definitions below).

Variables that refer functions are annotated the same way as function declarations, as described in the next section.

A variable that refer an object is annotated using `@type {Object}` doclet with a `@property` tag for each property. The doclet is associated with the variable name and the inferred type which is then marked as *exported*. An exception to this rule is if the type is already exported in which case a `@type {name}` doclet is generated instead with the name of the previous export of the type.

For object properties of primitive, simple and ⟨*jsd-union*⟩ types, the type expression is provided in the `@property` tag. Again, `void` types are not annotated.

For object properties of function or object types, the type is exported, if necessary, and the exported name is provided in the @property tag.

Annotation of Functions

Global functions are annotated using a ⟨*function-doclet*⟩ containing the tags `@param` and `@returns` as appropriate. The function declarations are identified statically as FunctionDeclaration nodes in the program body.

The type expression used in the `@param` or `@returns` tag is the same as for variables as described above. This includes generating virtual `@typedef` doclets as required to ensure that all referenced types are exported. The generated ⟨*function-doclet*⟩ it associated with the function type and it is marked as exported.

Function expressions are usually not named and even if they are, they must still be assigned to either a variable or object property to appear as an API member in JSDoc. If assigned to a top level variable, they are documented there using the same tags as for function declarations.

Annotation of Constructors

Constructor function annotations are the same as for regular functions, except that a `@constructor` tag is added to the doclet. Additionally, inside the constructor, assignments of the form `this.identifier` are annotated using ⟨*member-doclet*⟩ which JSDoc understands as class instance members. Prototype members are not inferred and thus not annotated either.

Virtual Type Definition Doclets

A ⟨*typedef-doclet*⟩ is generated when a ⟨*type*⟩ must be resolved to a ⟨*jsd-type*⟩ and contains a function or object type that is not exported. In this case, a ⟨*typedef-function-doclet*⟩ or ⟨*typedef-object-doclet*⟩ is generated for the non-exported type, assigned a global name and the type is exported using that name. The offending type is then replaced with a ⟨*jsd-reference*⟩ containing the exported name. Since ⟨*type*⟩'s can be recursive, resolving it implies recursively generating ⟨*typedef-doclet*⟩'s and exporting them as necessary.

The above implies that the order in which types are exported may influence how they are annotated. As an example, a function that is referred before it is declared, may cause an unnecessary ⟨*typedef-function-doclet*⟩ to be generated. This can be mitigated in an implementation by detecting the necessary exports and using a work list to optimize the export order.

Example

Listing 4.1 shows an example that uses all the described features.

Listing 4.1: Mode 1 example program with generated annotations

```
// Example A as an object

/**
 * @param {string} s
 * @returns {number}
 */
function fun(s) {
  return s.length;
}

/**
 * @type {Object}
 * @property {boolean} a
 */
var config = {a: false};

/**
 * @type {Object}
 * @property {number} b
 * @property {A_c} c
 * @property {fun} f
```

```
 * @property {config} config
 */
var A = {
    b: 5,
    c: {a: false, b: function(s){return false;} },
    f: fun,
    config: config
};

/**
 * @typedef {function} A_c_b
 * @param {string} s
 * @returns {boolean}
 */

/**
 * @typedef {Object} A_c
 * @property {boolean} a
 * @property {A_c_b} b
 */
```

Uninferable Tags

Some tags cannot be generated because they cannot be accurately inferred using dynamic analysis:

1. *Constants* It is detectable that a variable is only assigned a single value. However, inferring that such a variable is a constant is not practical because that would require testing all other variables with at least two different values. Thus, the `@const` tag is uninferable.
2. *Access modifiers* The access modifier tags, `@private`, `@protected` and so on are not inferable by the same argument as above.

4.3 Summary

The specification in this chapter allows generating JSDoc annotations that express most of the type information inferred in section 3.1.

Generating JSDoc annotations is challenging because is means solving two problems: Picking the right annotation based on the inferred type and locating the source symbol at which to attach it. There may not be a suitable source code symbol in which case a virtual doclet must be used instead. Additionally, the order in which annotations are generated is significant and an suboptimal generation order may result in unnecessary virtual doclets.

Chapter 5

Evaluation

5.1 Implementation

The inference tools is divided into in a number of modules that each implement its own area of functionality.

For parsing JavaScript code, traversing and manipulating AST's and subsequent code generation, the libraries Esprima[17], Estraverse[12] and Escodegen[11] are used. This family of tools build upon the Mozilla Parser API[8], discussed in the Language Tools section later in this chapter.

A class diagram of the implementation is shown in Figure 5.1.

Traverse Module

The traverse module contains the implementation for instrumentation and generation of JSDoc annotations as well as helper classes.

The `ScopeMap` class is used in AST visitors to assign identifiers to symbols and to track the AST nodes related to the symbols. An instance is generated during instrumentation and a matching instance must be used during type inference. For this purpose, the class supports serialization.

The `Instrumenter` object is an AST visitor that is invoked by Estraverse when traversing a syntax tree. It performs instrumentation by inserting instrumentation nodes into the AST during traversal. As the AST format is JSON, the nodes inserted are simply object literals containing values obtained by the `CodeToFromAst.html` tool discussed below.

`Annotator` is a simple class that transforms program code and annotations produced by `JSDocGenerator` into annotated code. This class is necessary because the comment support in Escodegen is not sufficient to generate annotations for all relevant statements and expressions (see the Language Tools section below).

The `JSDocGenerator` object is an AST visitor that translates inferred types into JSDoc annotations. The main algorithm for generating JSDoc annotations works as follows. Its input is

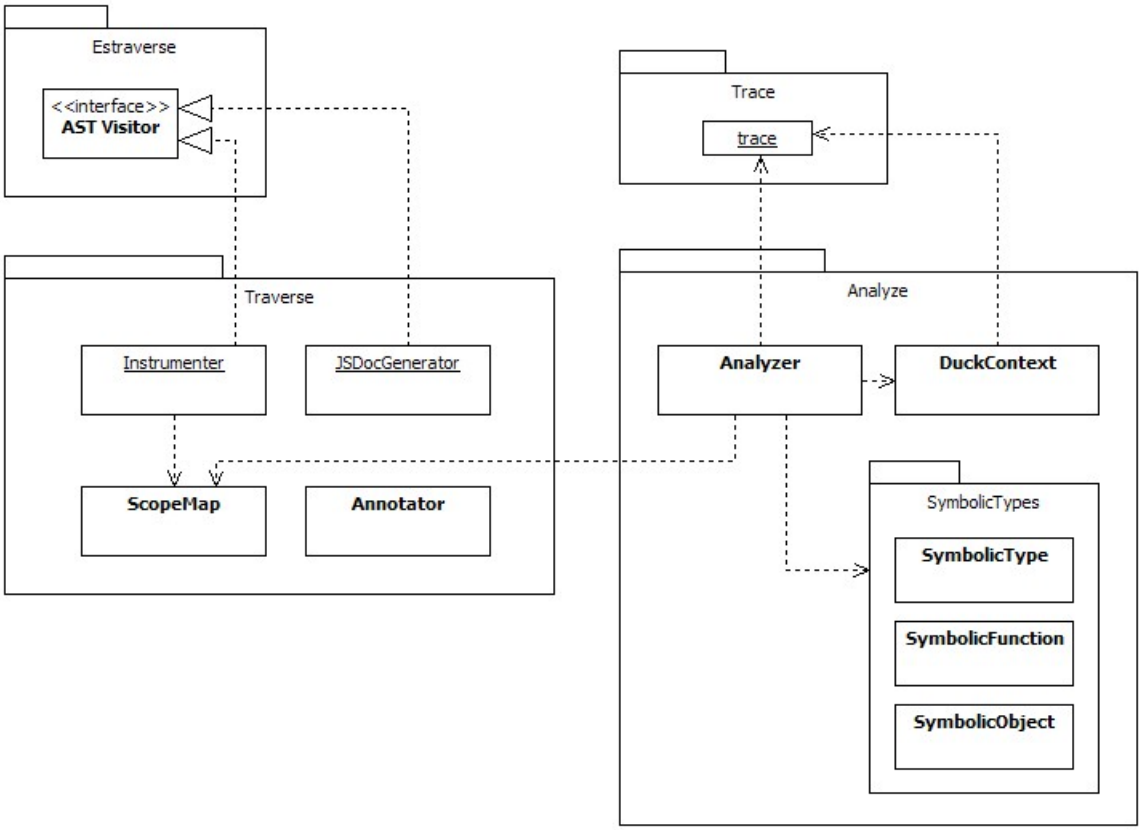

Figure 5.1: Class diagram of the implemented prototype. The entry points (Infertool.js and Analyze.html) refer all modules and are not shown.

a) an array of inferred types where global types have be augmented with its global name and *b)* a (uninstrumented) program AST augmented with source code locations for each node. The AST is traversed and *VariableDeclaration*s and *FunctionDeclaration*s in the program body are processed. The nodes are matched with its corresponding inferred type and an annotation object is created containing the appropriate doclet and location. Node processing includes recursively resolving and exporting dependent types.

The implementation is functional but contains the following issues that are postponed to future work:

- No effort has been made to optimize the export order. Thus unnecessary `@typedef` doclets may be generated.
- Naming of `@typedef` doclets is not guaranteed to be unique. A more sophisticated naming scheme could generate both unique and simpler names.
- Union types are not properly expanded in all cases because the naming scheme implemented generates very long names when all possible union type expansions are performed. This made testing difficult because the long names were difficult to predict.
- The properties of class types is obtained by inspecting a random class instance instead of deriving the intersection of properties of all instances as specified in section 3.1.

Trace Module

The trace module contains the methods that are called by instrumented code to produce trace log data. It also implements the wrapping of objects in a proxy for dynamic instrumentation.

The proxy implementation used is a *shim* called reflect.js[1] created by Tom Van Cutsem that implements the ECMAScript 6 Direct Proxy[2] specification. The shim is not required in Firefox which has a native implementation, but the V8 engine used in Chrome and Node.js requires the shim in addition to enabling ECMAScript 6 experimental features. During initial evaluation, the Firefox implementation was found to be significantly more stable than reflect.js. Some libraries caused the reflect.js to throw errors during the execution of instrumented code. In ECMAScript 6, the shim is no longer required which is likely to improve the situation.

The proxies are equipped with traps for property reads and writes that write the property key and value to the trace log. The proxy specification supports trapping functions as well (which are also technically objects) but functions were found easier to manage using static instrumentation.

Analyze Module

Type inference is implemented in the `Analyze` module. It uses the `SymbolicTypes` sub module to model the type system. Type inference takes as input the trace log and the `ScopeMap` created during instrumentation. The main functionality is implemented in the `inferTypes` method. It works by iterating over the symbols defined in the `ScopeMap` and, depending on the node type, calls the method responsible for inferring the symbol type.

Table 5.1 lists values and the method responsible for inferring their types.

[1] `https://github.com/tvcutsem/harmony-reflect`
[2] `http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies`

Type name	Method	Notes
Unknown values	`getValueType`	Inspects the value kind and delegates inference to the appropriate method.
Primitives and simple objects	`getPrototypeName`	Returns the type using `Object.prototype.toString.call()`.
Functions	`getFunctionType`	Inspects the trace log to infer the type. This may involve recursive calls to `getValueType`.
Objects	`getObjectType`	Inspects the trace log to infer properties and their types. This may involve recursive calls to `getValueType`.

Table 5.1: Type inference methods

The results are stored in a type map that maps symbol identifiers to inferred types. The inference methods use the classes `SymbolicType`, `SymbolicFunction` and `SymbolicObject` to model the types. They support operations such as building sparse union types, adding function parameters, setting functions as constructors and setting objects as classes. They all have a `toCanonicalString` method that builds the inferred type as a string.

The inference methods take a `context` parameter that is used in Duck Typing mode. This is a DuckContext instance that acts as a wrapper around the trace log, such that only trace data relevant in a certain context is iterated. When not in Duck Typing mode, there is only a single context instance that does not constrain iteration of the trace log. When in Duck Typing mode, a context is created for each function, which allows `getFunctionType` and its recursive calls to only consider trace log entries that occur inside that function. The DuckContext class uses a stack to track recursive functions.

In order to avoid (mutually) recursive types causing infinite recursion, a key is derived from the DuckContext which is used to detect and break the recursion by returning what corresponds to a `void` type. The same key is used to index types as a simple optimization to avoid processing already inferred types.

Tools

A number of tools have been developed as entry points to instrumentation, type inference and type annotation. The most important ones are:

Infertool.js A Node.js program for command line processing. Allows type inference, JSDoc generation and generation of instrumentation files.

Analyze.html A web application that provide interactive instrumentation, type inference and type annotation. This tool can be used to quickly test the system on a piece of code.

Instrument.html A web application used to visualize and debug instrumentation and assigned identifiers.

CodeToFromAst.html A web application that allows two-way translation of JavaScript code to AST. This tool was used to generate transformation code for instrumentation nodes.

Testing

The implementation is tested using the QUnit[3] testing framework. The test suite contains around 90 test cases.

Language Tools

AST Specification

In August 2010 the Mozilla SpiderMonkey JavaScript engine introduced a parser API, able to parse JavaScript source code and produce an AST in a specified JSON format[8]. This representation of a JavaScript AST has since become a popular format used by Esprima and Acorn, two JavaScript parsers written in JavaScript. So, while the native parser API is not available cross browser, two independent parsers are.

The format has been proposed[29] as an intermediate representation, shared by tools in order to produce composable JavaScript language tools.

An example of the format for the program `function foo (x) { return 1 }` is shown in listing 5.1.

Listing 5.1: Parser API AST example

```
{
  "type": "Program",
  "body": [
    {
      "type": "FunctionDeclaration",
      "id": {
        "type": "Identifier",
        "name": "foo"
      },
      "params": [
        {
          "type": "Identifier",
          "name": "x"
        }
      ],
      "defaults": [],
      "body": {
        "type": "BlockStatement",
        "body": [
          {
            "type": "ReturnStatement",
            "argument": {
              "type": "Literal",
              "value": 1,
              "raw": "1"
            }
          }
        ]
      },
      "rest": null,
      "generator": false,
      "expression": false
```

[3] http://qunitjs.com

```
      }
    ]
  }
```

Tools are available to traverse and manipulate this format as well as to generate back source code from it.

While the format is verbose it is easy to visualize and the entry level for producing language tools is lowered considerably compared to native code parsers and processors. As an example, a program for detecting leaked global variables can be implemented in 60 lines of JavaScript code[18].

CST Specification

Since the Parser API format represents an *abstract* syntax tree, some information from source code may be lost in a *code* $\rightarrow$ *AST* $\rightarrow$ *code* round trip, in particular comments and precise formatting. Such information is maintained in a *Concrete Syntax Tree* (CST). The esprima tools provide support for comments that allows, with some limitations, comments to be round tripped. It also supports various *location* information that specifies the source code location (i.e. line number and column) for nodes in the AST. This allows tools to do some processing of source code even though the AST format does not include all information from the original source.

However, including information such as comments and source location in the AST has inherent limitations due to the fact that some information cannot be expressed in the AST format because it lacks node types to represent it. An example is shown in listing 5.2.

Listing 5.2: Example of source code information unrepresentable in AST. Source: [10]

```
/*1*/ function /*2*/ foo /*3*/ ( /*4*/ ) /*5*/ {
  // ..
}
```

For example, the `/*4*/` comment occurs in an empty argument list which is represented in the AST as an array. Thus, when the arguments list is empty, no AST node is available on which to attach the comment. The esprima tools use 'extras' properties on existing nodes to represent comments, but not all information can easily be represented or processed this way. For example, 'extras' could include paranthesis that are redundant in the AST but would still need to be properly balanced when generating back code. This makes AST transformation potentially unsafe because a transformation would have to understand and process the 'extras' to avoid breaking a subsequent code generation.

These limitations have resulted in a very recent (March 2014) ad-hoc specification group[10] that aims to produce a CST specification for JavaScript. The purpose is to specify a format that can be used as an intermediate representation for tooling that requires more information than can be faithfully represented in the AST. Examples of such tools are source code refactorings in editors and JSDoc parsing and generation.

5.2 Experimental Evaluation

In the first part of this section, the type inference and annotation system is evaluated by applying the implemented prototype on two open source libraries. The actual libraries were chosen by considering whether an appropriate test suite was available and the difficulty in integrating the analysis system into the library test harness.

The second part presents a preliminary evaluation on the system applied to the jQuery library that mainly considers design and implementation robustness.

Research Questions

The evaluation technique is designed to answer the following research questions for a given library code base:

1. Is the type inference implementation able to accurately infer types for the library?
2. Is the JSDoc generation implementation able to produce correct annotations for the library?

In order to answer these questions the following measure is defined. It is used to separately measure type inference accuracy and annotation correctness.

Evaluation Measure

The evaluation is performed by applying the implemented prototype on each library. For each library API member, the generated result is assessed and scored from one to five, once for type inference and once for JSDoc annotation. The score is derived using the following measure:

> **5** is the best score. It indicates that the inferred type or the JSDoc annotation accurately captures the intention of the API member.
>
> **4** indicates the the result closely matches the member intention, but is not completely accurate because the optionality of (part of) a type is incorrect.
>
> **3** indicates that the member intention was captured but not precisely expressed. This is the case if part of a type was not inferred or an inaccurate JSDoc annotation was applied.
>
> **2** indicates that the member intention was not captured because no type or the wrong type was inferred. For annotations, it signifies that an inappropriate or no annotation was applied.
>
> **1** indicated a that no result could be achieved because the inference system could not be run against the library.

In addition to the above measure, a qualitative assessment is made of the evaluation result as a whole.

Evaluation: Accounting.js

Accounting.js[4] is *a tiny JavaScript library for number, money and currency formatting, with optional excel-style column rendering (to line up symbols and decimals). It's lightweight, fully localisable and has zero dependencies.*

The library API consists of just 5 public methods implemented in 412 lines of code. In addition to these function members, a settings object is described in the documentation. This part was ignored in the analysis, since it is largely prose documentation and not type oriented.

The library does not come with a test suite, but a small number of tests were available in the DefinitelyTyped[7] repository. These tests were generated by the author of the TypeScript Type Definition files for the library for testing the definitions.

Analysis Technique

The library was evaluated using the `Analyze.html` program in Firefox by inserting the library code and test code. The Analyze button was used to generate the annotations. Then, by opening the JavaScript console and clicking Log signatures, the inferred types were inspected, cross referenced with the assigned ids which were obtained using the `Instrument.html` program.

Type Inference

All member types were faithfully captured except optionality in some cases, thus all yielding an initial score of 4. The missing optionality was caused in part by missing test cases and in part by heavily overloaded members. The overloaded members make the result somewhat hard to read, as seen in the following example. The function is overloaded by allowing the `symbol` parameter to contain an object literal instead of providing the remaining parameters.

Listing 5.3: Inferred type for the formatMoney function.

```
//Function signature
var formatMoney = function (number, symbol, precision, thousand, decimal, format)
    {...}

// Inferred type:
fn(
  number:(number|Array),
  symbol:(
    string|
    {
      symbol:string,
      format:string,
      decimal:(undefined|string),
      thousand:(undefined|string),
      precision:(undefined|number),
      grouping:(undefined|number)
    }|
    {
      symbol:string,
      precision:number,
```

[4] http://josscrowcroft.github.io/accounting.js/

```
        thousand:string,
        format:{pos:string, neg:string, zero:string, charCodeAt:undefined},
        decimal:(undefined|string),
        grouping:(undefined|number)
      }
    ),
    precision:number,
    thousand:(string|undefined),
    decimal:(string|undefined),
    format:undefined
  )
    -> (string|Array)
```

The TypeScript type definition file for the library provides much better results because TypeScript supports method overloading, provided that care is taken to define the overloads in a supported way. This may not be possible for all API's and in any case it is a manual design effort.

Annotations

Because the library uses a module pattern wrapper (see section 2.1), the source code was modified by removing the wrapper prior to analysis. This allowed the system to annotate 4 of the 5 members. The remaining member was not annotated because it was not exposed in a supported way. The generated annotations mimicked the inferred types and were given a score of 4 while the 5th member was scored 2.

Overall, the style of module code is not properly supported which means that the result has to be modified by hand to produce a well formed JSDoc annotated program.

Conclusion

The conclusion for the Accounting.js library is that types were fairly accurately inferred with an average score of 4.

The generated member annotations were somewhat complete with a average score of 3.6, albeit with a missing `@module` doclet. An annotation score of 4 could be obtained by supporting the module pattern used which would also alleviate the need for hand editing the result.

Evaluation: Js-Signals

Js-signals[5] is a *Custom Event/Messaging system for JavaScript inspired by AS3-Signals.*

The library is implemented in 445 lines of code and consists two classes, `Signal` and `SignalBinding`. The library API consists of 18 methods that are exposed by the two classes.

The library test suite contains 49 test cases implemented using the Jasmine[6] test framework.

[5] `http://millermedeiros.github.io/js-signals/`
[6] `http://jasmine.github.io/`

Analysis Technique

The library was instrumented using the `Infertool.js` script which also generates a serialized `ScopeMap` file. The test runner html page was modified to load the instrumented library code and scope map. Finally, analysis was run by injecting analysis code into the test runner html page to run upon test suite completion.

Type Inference

Types were inferred for all 18 API methods. The methods and their scores are listed in table 5.2.

Method	Score	Notes
`Signal.add`	3	*
`Signal.addOnce`	3	*
`Signal.dispatch`	4	**
`Signal.dispose`	5	
`Signal.forget`	5	
`Signal.getNumListeners`	5	
`Signal.halt`	5	
`Signal.has`	5	
`Signal.remove`	3	*
`Signal.removeAll`	5	
`Signal.toString`	2	***
`SignalBinding.execute`	4	****
`SignalBinding.detach`	5	
`SignalBinding.isBound`	5	
`SignalBinding.isOnce`	5	
`SignalBinding.getListener`	5	
`SignalBinding.getSignal`	2	*****
`SignalBinding.toString`	2	***

Table 5.2: Evaluation of js-signal API members

Table 5.2 notes:

* The method contains error handling which was tested by supplying parameters of invalid types. This resulted in the erroneous parameter types contributing to the inferred parameter types.

** The method has a very loose signature allowing any number of parameters of any types to be sent to registered listeners. This was only partially reflected in the inferred type.

*** The methods was not tested by the test suite. They are considered part of the API since they are listed in the documentation, though the missing tests indicate otherwise.

**** The return value depends on the registered listener and is thus arbitrary. The inferred function return value does not completely reflect this.

***** An exception occurred which caused the constructor function to not be inferred. Therefore the return value inferred did not have the intended `Signal` type.

Annotations

Annotation generation was run, but no annotations were generated for the library because the coding style includes both modules and namespaces and all public members are implemented as prototype properties. As all three of these elements are not supported by the implementation, merely removing the module wrapper as was done for the previous library would not be sufficient. This yields a score of 2 for all 18 members.

Conclusion

For Js-Signals, type inference was fairly accurate with an average score of 4.1. The result could be improved by supporting prototype properties.

No annotations could be generated yielding a average score of 2. A score of 4 could be likely obtained if the annotation generation supported modules and prototype properties, even if namespace support was not implemented. This would just cause the namespace to be annotated as an object.

Preliminary Evaluation Of jQuery

The popular jQuery library is a challenging evaluation case. It is challenging in part because it is large code base consisting of 9190 lines of unminified JavaScript code. Its test suite consists of 6218 asserts in 806 test cases. It is also challenging because of its diverse functionality, including DOM manipulation which can only be run inside the browser where these manipulations can be performed.

Table 5.3 provides an overview of the functional areas of jQuery's API and information about its members.

Area	API members
Ajax	18
Attributes	10
Callbacks	12
Core	5
CSS	16
Data	10
Deferred	19
Effects	19
Events	58
Manipulation	58
Traversing	29

Table 5.3: Functional areas in jQuery. The list was compiled from API documentation. Some areas were removed because their members were present in multiple areas.

Research Questions

Because of the large number of API members, an evaluation as given in the previous sections is an unwieldy task. Instead, the primary concern is evaluating the robustness of the prototype design and implementation when subjected to the diverse jQuery tests.

Thus, the research questions for this preliminary evaluation is:

1. Is the instrumentation design and implementation robust enough to run in the jQuery test suite without generating errors of its own?
2. If errors occur, are they design or implementation errors in the prototype or are they errors in the jQuery implementation or test suite?

Analysis Technique

Running the analysis involves these steps:

Instrumentation. This was performed using the `Infertool.js` script which ran without problems. Instrumenting the library resulted in a 311 KB file vs. the original size of 242 KB (83 KB minified).

Execution. The jQuery test harness is based on the QUnit[9] test framework, a single html page hosting the test runner and a number of PHP scripts that respond to AJAX requests during testing. Integrating the prototype into the test harness was done by changing the jQuery library reference to the instrumented version and including the prototype script references in the html page. This allowed the tests to run while generating trace log data based on the test cases. The test cases themselves were not instrumented because subsequent instrumentation of multiple files has not been implemented. Thus it would require significant work to instrument the tests. The missing instrumentation means that the types of functions (typically callbacks) and objects generated in the tests were not inferred.

The jQuery test suite contains 806 tests totalling 6218 assertions. On the test machine, the suite ran in 50 seconds on the original code. When switching to the instrumented code the suite took 602 seconds, an increase of 12X. Running the test cases generated 1,9 million (1876066) trace log entries. On the surface, a 12X increase in running time looks good in comparison with the Jalangi framework that reports an average time increase of 26X during recording. However, any possible improvement cannot be determined based on this measurement because the jQuery tests perform time consuming DOM manipulation and AJAX requests, part of which are not influenced by instrumentation.

Running the test suite on the instrumented code caused a significant number of failed tests. Since instrumentation should not affect the external behaviour or the code this could indicate problems in the instrumentation design or the implementation. Instead of 6218 assertions only 5396 were run of which 658 failed. A partial analysis of the failing tests indicate the following primary causes:

- The instrumentation properties `__id` and `__class` that are added to proxy objects is the cause of most of the failing assertions. Some tests fail because they assert only certain properties to exist, an assertion which is violated by the instrumentation. Other test fail because part of the jQuery implementation does not expect these properties to exist. This

appears to be a known problem[7] with the `Sizzle` framework used by jQuery which fails if any properties are added to `Object.prototype`. The instrumentation properties is also the reason that most Ajax tests fail (284 of 329 assertions) and Data (57 of 391 assertions).

- Dynamically generated `iframe` elements. Instrumented code that runs inside generated iframes does not have access to the trace methods and thus throw an exception when called. This causes tests to fail, mostly by timing out because an expected callback function is not called.

Type Inference. Type inference was run by adding a script to the test runner page. Type inference took 34 minutes generating a total of 820 inferred types. The correctness of the inferred types was not investigated beyond simple smoke tests to verify that analysis was run to completion.

Conclusion

The implemented prototype was able to run in the jQuery test suite but did produce a number of errors. The primary cause of the errors was found to be that jQuery tests and implementation does not support the instrumentation properties `__id` and `__class`. This can be seen either as a design flaw in the prototype or as a flaw in jQuery tests and implementation.

The `iframe` related failures is an implementation flaw or missing feature in the prototype. To fix it, plumbing code could be inserted into generated iframes, such that trace calls were delegated to the containing window. This should be possible as the generated iframes are located on the same domain, which means that the plumbing code would not be subject to *Same Origin Policy* restrictions.

[7] `https://groups.google.com/forum/#!topic/jquery-dev/0-CEDwnGDlg`

5.3 Related Work

There is a large body of work related to JavaScript type inference and other analyses.

Jalangi[28] by Koushik Sen et. al is a selective record-replay and dynamic analysis framework for JavaScript. It uses comprehensive instrumentation of JavaScript source code to allow recording and replaying of parts of a program execution. It also supports shadow execution to enable replay of code that access native or remote resources without access to such resources during replay. The framework is intended for general purpose analyses and specific analyses presented include concolic testing, taint analysis and detection of probable type errors. The dynamic analysis instrumentation technique used in this paper is derived from the Jalangi system, but introduces proxies to dynamically trace object property reads and writes. The Jalangi system was considered when implementing the type inference prototype, but a hybrid instrumentation designed specifically for type inference was found to be a simpler approach.

In [2] An Jong-hoon et. al present Rubydust, a constraint based type inference system for Ruby based on dynamic analysis. The technique uses introspection to wrap run-time values and associate them with type variables. The wrapper generates subtyping constraints for the type as the wrapped value is used, and types are inferred by solving the constraints. They note that in order to infer sound method types, all code paths in methods must be executed and prove a soundness theorem for a small Ruby language subset. A similar soundness theorem does not hold true for the system presented in this paper. Test cases are suggested for training the system and type annotations provided by the programmer are also supported. Rubydust uses dynamic instrumentation by patching the input program at run-time using Ruby's built-in introspection features. The technique presented in this paper uses source code instrumentation combined with dynamic patching of object instantiations. The analysis method of constraint generation and solving differ from the method in this paper that uses recursive union types as the type system model. The constraint model infers more general types than the method described here, although the Duck Type inference mode presented in section 3.1 likely produces similar results in many cases. As Ruby supports classes, the target of type inference in [2] is chosen to be class methods and fields. Since JavaScript does not support classes natively, the scope of JavaScript type inference is slightly larger as both variables, functions, objects, methods and classes must be supported in order to handle real-world code.

TypeScript[25] is a superset of JavaScript, developed by Microsoft in 2012. Its main features are language support for optional static typing, classes, interfaces, modules and generics. It consists of a language specification[24] and a compiler that compiles TypeScript into ECMAScript 3. Additionally support for TypeScript is available as plug-ins for a number of editors. Being a superset of JavaScript, any JavaScript code can be consumed from TypeScript. Via an annotation format named Type Definition files, existing JavaScript libraries can be annotated with type information which can be used by the compiler for type inference and type checking of client code. The features of its annotation format are highly similar to JSDoc with the major difference that type information is part of TypeScript syntax and documentation are just comments, whereas in JSDoc all information is included in specially formatted comments. The structural type system presented in this paper is similar to that of TypeScript. A significant difference is that in the former, functions and objects are distinct, while in TypeScript a function is an object equipped with one or more call signatures. While the TypeScript approach is more general, the function/object distinction was chosen because it is prevalent in practice like in most other languages. The choice of being a superset of JavaScript sets TypeScript apart and makes it a large and complex language. Most other approaches instead try to limit language features by

excluding the problematic features (see section 2.1). Examples of this are CoffeeScript[8] and Dart[9].

In [3], Anderson et. al presents a static analysis method for type inference of an idealized subset of the JavaScript language. They define a the language with a flexible type system that allows objects to have properties dynamically added. A typed version of the language is defined and a sound type inference algorithm is presented that, in case all type constraints are solvable, is able to translate programs to the typed language.

In [21], Simon Holm Jensen et. al present a static analysis framework for the full ECMAScript 3 language that is based on abstract interpretation. The aim is to infer sound and detailed type information that can be used for program verification and interactive editor warnings and code completion hints. The method supports `eval` by translating such calls during flow analysis.

The static analysis approaches to type inference can be seen as an over-approximation whereas dynamic analysis using test cases is an under-approximation. As discussed in section 3.3, dynamic analysis tends to perform well when targeting semantically focused API's whereas static analysis considering most or all possible execution paths needs some degree of implementation structure to infer results. Conversely, soundness can often be guaranteed using static methods whereas the system presented here is both theoretically and practically unsound.

In [13], Feldthaus et. al present a static analysis technique for efficient JavaScript call graph construction. This work is aimed at interactive code assist services in editors, such as *Go to declaration* and automatic refactorings. Their work aims to tackle the performance requirements of interactive use cases, by employing an approximate field-based flow analysis that scales and performs well. In a similar vein, The Gatekeeper[15] system by Guarnieri et. al, defines the 'points-to' relation in order to specify security constraints for embedding 3rd party JavaScript widgets. By introducing a statically analysable JavaScript subset and using dynamic analysis to further expand the domain of analysable programs, the 'points-to' relation can be mostly statically inferred. These approaches both provide solutions to concrete problems that in strongly typed languages are generally solvable in a performant way by utilizing the information and guarantees provided by the type system. As such, they illustrate the difficulties statical analysis face when analysing dynamic languages.

[8] `http://coffeescript.org/`
[9] `https://www.dartlang.org/`

Chapter 6

Conclusion

A type inference system that uses dynamic analysis based on test cases has been specified. The system has been implemented in a prototype and evaluated using experimental evaluation. The system was found to provide useful and accurate results in many cases using existing test cases for the evaluated libraries.

A translation has been specified that translates inferred types into JSDoc type annotations. Together with the type inference system, this translation is found to be useful to automatically generate type annotations for existing code. While the generated annotations are not complete in the evaluated cases, they provide a useful starting point and eliminate much of the grunt work required to produce and maintain up-to-date type annotations.

6.1 Future Work

The presented system can be improved in a number of areas:

The type inference specification can be extended to support prototype properties, sub-classes and tracking of the *this* pointer. These constructs are easily inferable using dynamic analysis. Support for `eval` can be added by runtime-instrumentation of string arguments passes to the function.

Generation of JSDoc annotation can be improved by supporting additional kinds of doclets, especially modules that are used heavily in modern libraries. This requires additional instrumentation and additions to the type inference specification. The current implementation can be improved by a more sophisticated export order and better naming generation scheme.

Additional type annotation formats could be generated. An obvious choice is TypeScript Type Definitions. This would allow using the TypeScript compiler to perform type checking of client code based on generated type definitions. While the TypeScript language is very large, it is believed that a suitable subset could be selected to match the type inference specification. Generating TypeScript type definitions may even be simpler than JSDoc generation because TypeScript type definitions are disjoint from the original JavaScript source code.

A more thorough evaluation of the system using multiple benchmarks would likely point to areas where both robustness and accuracy could be improved. Additionally type inference in 'Duck

Typing mode' could be evaluated and compared with results in 'normal' mode.

The system could support incremental inference and annotation, where trusted type annotations are supplied by the user and the remaining types are inferred and annotated. This would allow the user to correct incomplete annotations (for generic methods for example) while still using the system moving forward.

The instrumentation mechanism might be implemented entirely dynamically using introspection similar to the Rubydust[2] system. The Proxy specification supports most of the features required for this.

Bibliography

[1] Ben Alman. Immediately-invoked function expression (iife). http://benalman.com/news/2010/11/immediately-invoked-function-expression/, November 2010. Accessed May. 5th 2014.

[2] Jong-Hoon AN, Avik CHAUDHURI, Jeffrey S FOSTER, and Michael HICKS. Dynamic inference of static types for ruby. *ACM SIGPLAN notices*, 46(1):459–471, 2011.

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *ECOOP 2005-Object-Oriented Programming*, pages 428–452. Springer, 2005.

[4] Ben Cherry. Javascript module pattern: In-depth. http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html, March 2010. Accessed Feb. 19th 2014.

[5] Jim Chow and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008.

[6] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., first edition, May 2008.

[7] Boris Yankov et. al. Definitelytyped, the repository for high quality typescript type definitions. https://github.com/borisyankov/DefinitelyTyped. Accessed Feb. 9th 2014.

[8] Dave Herman et. al. Parser api. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API, June 2010. Accessed Apr. 3rd 2014.

[9] John Resig et. al. Qunit, javascript unit testing framework. http://qunitjs.com. Accessed Feb. 9th 2014.

[10] Kyle Simpson et. al. Concrete syntax tree. https://github.com/getify/concrete-syntax-tree, March 2014. Accessed Apr. 3rd 2014.

[11] Yusuke Suzuki et. al. Escodegen (escodegen) is ecmascript code generator from parser api ast. https://github.com/Constellation/escodegen. Accessed Feb. 9th 2014.

[12] Yusuke Suzuki et. al. Estraverse (estraverse) is ecmascript traversal functions from esmangle project. https://github.com/Constellation/estraverse. Accessed Feb. 9th 2014.

[13] Asger Feldthaus, Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 752–761. IEEE, 2013.

[14] Google. Closure compiler. https://developers.google.com/closure/compiler/. Accessed Mar. 8th 2014.

[15] Salvatore Guarnieri and V Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[16] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *ACM SIGPLAN Notices*, volume 47, pages 239–250. ACM, 2012.

[17] Ariya Hidayat. Ecmascript parsing infrastructure for multipurpose analysis. http://esprima.org. Accessed Feb. 9th 2014.

[18] Toby Ho. Fun with esprima and static analysis. http://tobyho.com/2013/12/02/fun-with-esprima/, December 2013. Accessed Apr. 3rd 2014.

[19] Ecma International. Ecmascript® language specification. http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf/, June 2011. Accessed May. 15th 2014.

[20] Matthew Kastor et. al Jade Dominguez. Official jsdoc website. http://usejsdoc.org/. Accessed Mar. 1st 2014.

[21] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis*, pages 238–255. Springer, 2009.

[22] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, 44(6):110–120, 2009.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[24] Microsoft. Typescript language specification. http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf. Accessed Apr. 21st 2014.

[25] Microsoft. Typescript language web site. http://typescriptlang.org/. Accessed Mar. 11th 2014.

[26] Eric Miraglia. A javascript module pattern. http://yuiblog.com/blog/2007/06/12/module-pattern, June 2007. Accessed Feb. 19th 2014.

[27] Dr. Axel Rauschmayer. *Speaking JavaScript*. O'Reilly Media, Inc., first edition, March 2014.

[28] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 615–618. ACM, 2013.

[29] Yusuke Suzuki. Escodegen and esmangle: Using mozilla javascript ast as an ir. http://aosd.net/2013/escodegen.html, March 2013. Accessed Apr. 3rd 2014.