JSTrace: Run-time Type Discovery for JavaScript

Claudiu Saftoiu

May 13, 2010

Contents

Chapter 1

Overview

Web programming is in a state of turmoil. Developers must use technology unsuitable for the complex requirements of modern Web sites. As new programs serving a widening array of purposes are increasingly deployed on the Web, the situation will only worsen.

Modern Web sites, such as Gmail and Facebook, are powered by the programming language JavaScript. Even relatively static sites rely on JavaScript to function, such as the home page of the New York Times, which uses numerous program elements to display movies, ads, picture galleries, and so forth. Unfortunately, JavaScript has many warts that make it a poor choice to use to develop large-scale programs.

JavaScript poses challenges that are new to programmers used to developing traditional desktop programs. For example, one issue Web developers must deal with is incompatibility between browsers. A given JavaScript program might function flawlessly in one browser, but be severely broken when viewed with others. Creating a Web application that can run in all browsers is not trivial, as evidenced by many websites that require a particular browser to be viewed. JavaScript's evolution as a language is to blame. It was created without a formal specification and implemented by several browser makers who did not collaborate to ensure consistency amongst their implementations.

Security is another vital issue. It is exceedingly easy to unintentionally create JavaScript programs containing security vulnerabilities. Symantec, a maker of anti-virus software, documents 11,253 occurrences of a particular type of vulnerability over a six-month period [7]. Those with malicious intent have reacted accordingly, proliferating Web programs exploiting these vulnerabilities. For perspective, Google estimates that about one in ten Websites could contain malicious code [4]. Symantec admonishes users to turn off JavaScript support whenever visiting untrusted sites [8], which most users do not know how to do. The cause is once again related to JavaScript's development. The language was designed fifteen years go, without the concerns of modern Web applications in mind. As the scope of Web sites has expanded to fit today's broad needs, JavaScript's fundamental design has remained unchanged.

To begin to improve the state of Web programming, we must first deal with JavaScript. Given the language's many shortcomings, we might be tempted to replace JavaScript with a technically

Program	Lines of Code	Overhead (Annotated LOC)
watchimer	947	1.80%
countdown	129	3.10%
topten	443	4.06%
morse	275	4.36%
watchimer	947	4.44%
resistor	591	5.41%
catchit	165	5.45%
rsi	328	6.71%
animation	70	7.14%
hashapass	257	7.78%
text2wav	488	8.40%
metronome	106	11.32%
TOTAL	3799	5.05% (avg)

Figure 1.1: Annotation overhead on JavaScript code

superior language. Unfortunately, JavaScript is pervasive; one estimate shows that 73% of websites rely on JavaScript for important functionality [3], while a review I performed of the top 500 visited sites on the Internet[1] revealed that all 483 sites that I was able to access at the time used JavaScript. A solution replacing JavaScript will break compatibility with existing Web infrastructure.

Instead, we need to give Web programmers tools and techniques that enable them to reason about existing JavaScript code. Along with my advisors, I have developed *Typed JavaScript* [14], a tool that fits this description. Typed JavaScript is a *type system*, which is a system that allows programmers to document the behavior of their code using a formal language called *types*. A *type checker* then automatically **proves** that the documentation is correct. Type systems make programs easier to manage and helps assure developers that they are correctly reasoning about their code. Developers already use type systems in the field, as many industrial languages such as Java and C# use simple versions of them.

To use Typed JavaScript, programmers merely have to insert annotations that document their program's expected behavior. However, even this can be a burden! Our empirical evidence (figure 1.1) shows that adding annotations entails documenting 1-12% of a program. This can be a laborious process, as successfully annotating a program requires understanding and keeping large parts of the code in mind simultaneously. The overhead of adding annotations limits Typed JavaScript's usability, as Web developers must take a non-trifling amounts of time to change their code to work with our type system.

My independent contribution is JSTrace, a tool that provides a method of automatically annotating existing code. My tool directly tackles the burden of annotating JavaScript code, making Typed JavaScript a more effective tool. By facilitating the process of using a type system to reason about Web programs, JSTrace directly confronts issues plaguing Web developers today, such as browser incompatibility and Web security.

[1] according to Alexa Internet, a company that tracks website usage information.

In collaboration with my advisors, I have made the following contributions:

- We created and published a *semantics* [13] for JavaScript. A semantics is a formal description of how programs in a given language function, and gives insight into the nature of a programming language.
- We evaluated our semantics on a significant portion of a browser test suite, showing that the semantics corresponds to reality.
- We used our semantics to build a safe subset of JavaScript that prevents a particular type of security vulnerability.
- We conducted a survey of real-world JavaScript programs, focusing on idioms that employ JavaScript's quirks.
- We developed a type system for JavaScript that handles these quirks, allowing developers to use modern technology to reason about their programs.

In this document, I present JSTrace. I first discuss the general idea behind my tool and choices made when developing it. I then expose the technical details of the tool's functionality. In the evaluation chapter, I demonstrate that the tool is effective at reducing the burden of typing programs. I then conclude, discuss future work, and review related work.

Chapter 2

Introduction

Thesis Statement Run-time analysis on JavaScript is an effective way to reduce the burden of porting existing JavaScript code to Typed JavaScript.

2.1 Static Type Inference

Languages such as Haskell and OCaml use static type inference to reduce the amount of annotations required. Type inference is sound, and provides the principal type for any expression [21]. We might try to apply this technique to Typed JavaScript.

However, Typed JavaScript's type system is not conventional. One of the goals of Typed JavaScript is to be able to type existing code with as little overhead as possible. Annotations are not the only type of overhead, however. Code from languages without a type system, such as Python and JavaScript, often looks very different than code from a statically typed language, such as Java. It uses idioms that traditional type systems would reject. Thus, if Typed JavaScript were designed as a conventional type system, we would either have to reject programs using these idioms, or require the idioms to be refactored to fit our type system. This is undesirable, so Typed JavaScript is engineered to admit as many of these idioms as possible.

2.1.1 Subtyping

JavaScript programs employ both nominal and structural subtyping. As a result, the type system supports both. An example of nominal typing is in the browser DOM, where `HTMLDocument` is a subtype of `Node`. An example of structural typing is with structural object types — the type `{x : Int, y : Int}` is a subtype of the type `{x : Int}`.

The traditional Hindley-Milner type inference algorithm doesn't work with subtyping. Variants have been developed that do work with subtyping [22]. However, JavaScript has other properties that make static inference difficult.

```
function double(a) /*: Int + {val : Int} -> Int */ {
  if (typeof a === "number")
    return a * 2;
  else
    return a.val * 2;
}
double(4);
double({val: 15});
```

Figure 2.1: JavaScript code exhibiting flow-based reasoning

2.1.2 Flow-based Reasoning

JavaScript programs use control flow to reason about types. For example, the function `double` in figure 2.1 works on both integers and on objects containing an integer field, `val`. It uses a primitive JavaScript operator, `typeof`, to branch on the type of the argument it was supplied. The type of the argument is the type union `Int + {val : Int}`.

A traditional type checker would fail on this code for two reasons. It would not allow `a * 2`, because `a` might be an object type, and it would not allow `a.val * 2`, because `a` might be an integer. However, Typed JavaScript takes control flow into account using flow analysis. The conditional ensures that in the true branch of the if statement, `a` is an integer, and in the false branch, an object. The comment in the figure is a type annotation, and the code snippet is valid Typed JavaScript code. A static type inference algorithm would also have to take these flows into account.

2.1.3 Ambiguous Syntax

In addition to complications with the type system, JavaScript's syntax and semantics itself poses problems. A static inference algorithm must, by definition, work over syntax. To complicate matters, JavaScript's syntax is ambiguous. For example, the expression `foo[idx]` could either be:

- Array access:

  ```
  var foo = [1, 2, 3, 4];
  var idx = 3;
  ```

  In this case, `foo` is an array, and `idx` indexes into the array.

- Object property access:

  ```
  var foo = {Bob: 20, Jill: 30};
  var idx = "Jill";
  ```

  In this case, `foo` is an object used as a mapping between strings and integers. `idx` is a string.

The same goes for dot references. For example, `foo.length` could mean:

- Getting the length of an array: `var foo = [1, 2, 3, 4];`
- Getting the length field of an object: `var foo = {length: "long"};`.

It is difficult to statically distinguish functions and constructors. Examine the following function:

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

Based on the function's call sites, it can be:

- A constructor:

```
var pt = new foo(10, 10);
```

- A method:

```
var obj = {x: 0, y: 0, set: foo};
obj.set(10, 10);
```

- Even a method working on global variables:

```
var x = 3, y = 5;
foo(10, 10); //sets the global variables x and y!
```

Furthermore, JavaScript operators perform rife amounts of implicit type conversions. The expression `a * b` can be valid whether its operands are:

- Numbers:

```
var a = 3, b = 5;
```

- Strings, which are implicitly converted to numbers:

```
var a = "3", b = "5";
```

- Objects, which are implicitly converted to numbers:

```
var a = {valueOf: function() { return 3; }};
var b = {valueOf: function() { return 5; }};
```

- Objects, converted to strings, and then converted to numbers:

```
var a = {valueOf: function() { return "3"; }};
var b = {valueOf: function() { return "5"; }};
```

- Any combination of the previous!

```
var a = {valueOf: function() { return "3"; }};
var b = 3.4;
```

Even when syntactically referencing the property of an object, e.g. `num.toString()`, the operand is not guaranteed to be an object. If `num` is the value `42`, for example, the integer is implicitly converted to an object whose `toString` method returns `"42"`.

These properties of JavaScript make it difficult to infer types soundly. However, it's important to note that type checking and type inference are two separate algorithms. The Hindley-Milner

algorithm is particularly desirable, since it is sound and infers the principal type, but this need not be the case. As long as we have a sound type checker, we can run any inferred types through that type checker to verify them for correctness.

With this in mind, we can attempt to reduce the annotation burden.

2.2 Variable Annotations

Languages like C, C++, Java and, until recently, C#, require annotations on variable bindings as well as functions. In Typed JavaScript, this is optional. By default, when the type checker sees a variable binding, it takes the type of the variable to be the type of the expression initializing it. For example, the following is valid Typed JavaScript code, giving the variables types `Int`, `Boolean`, and `{x : Int, y : Int}` respectively:

```
var a = 0;
var b = true;
var c = {x: 29, y: 30};
```

This type is not always correct. For example, if a variable has some union type, and it is initialized with a value having only one of the types, then the correct union type will not be calculated. In these cases, the type checker returns an error, and a type annotation must be added.

Unfortunately, function annotations cannot be calculated. Unlike variable bindings, where the initial value provides a good hint as to the variable's type, a function's type depends on its entire body, as well as all its call sites. In the next section, I present a tool which effectively discovers function annotations using run-time analysis.

Chapter 3

JSTrace

I have developed JSTrace, a run-time analyzer to discover Typed JavaScript types. JSTrace has two components: a JavaScript file implementing the functionality of the tool, and a compiler that inserts the hooks into a JavaScript program necessary for the tool to work.

3.1 Overview

Since the overhead of annotating local variables is mostly gone, JSTrace's only goal is to discover and output the types of the functions in a program. To use the tool:

- Compile the file to be traced. This wraps expressions, such as functions, as necessary for the tool to be able to track type information.
- Run the compiled code and explore as many code paths as possible. As the compiled code runs, more and more accurate information is gathered and outputted to the user.
- Use the output to automatically annotate the original source code.
- Run the annotated code through the type checker to verify the types.

Upon wrapping a function, we have already discovered an arrow type. At this point, we have no information about what types it takes or returns. This is indicated in figure 3.1(a) with `Unknown`. On every invocation of the function, we can inspect the values given as parameters and returned from the function at run-time (figure 3.1(b)). When new invocations reveal values of different types, we discover a type union (figure 3.1(c)).

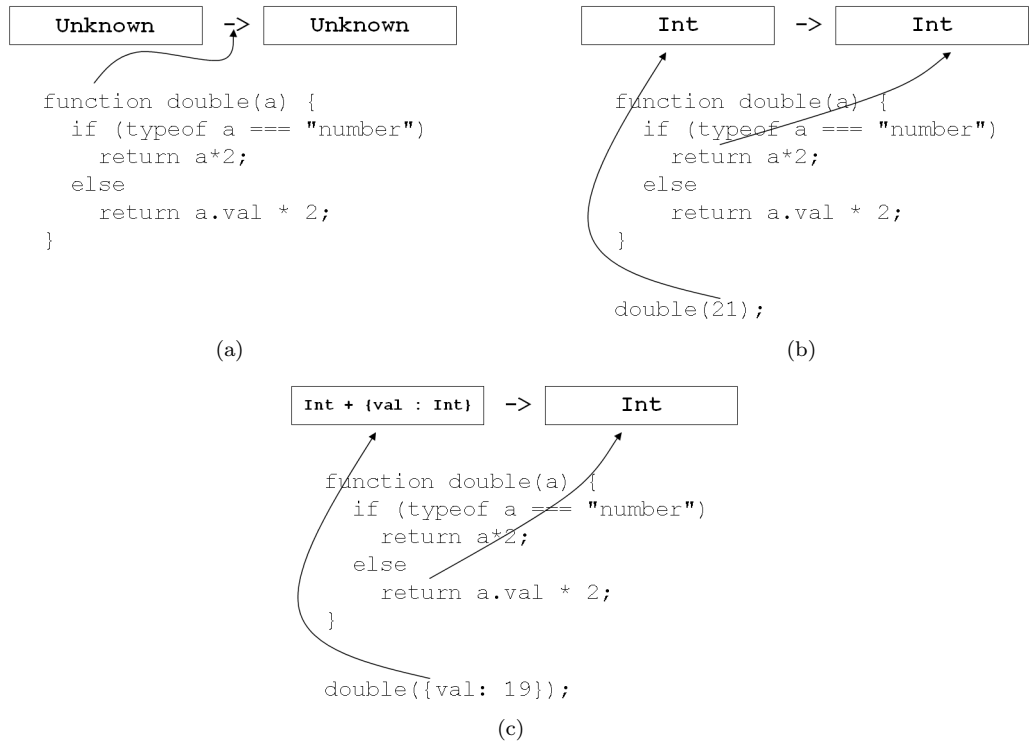

Figure 3.1: Discovering function types

$$
\begin{array}{rcl}
unk & = & \texttt{Unknown} \mid \texttt{Host} \mid \texttt{UntracedFunction} \\
flat & = & \texttt{undefined} \mid \texttt{null} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{double} \\
builtin & = & \texttt{Date} \mid \texttt{RegExp} \mid \texttt{Array<}tag\texttt{>} \\
func & = & \texttt{[}tag\texttt{]}\ tag\ \cdots\ \texttt{->}\ tag \mid \texttt{constructor}\ id : tag\ \cdots\ \texttt{->}\ tag \\
obj & = & \texttt{\{}\ id\texttt{:}\ tag\ \cdots\ \texttt{\}} \mid \texttt{InstanceOf<}id\texttt{>} \\
tag & = & unk \mid flat \mid builtin \mid func \mid obj \mid tag + tag \mid \texttt{Ref<}id\texttt{>}
\end{array}
$$

Figure 3.2: JSTrace tag language

3.2 Tags

JavaScript code is inherently untyped. It's technically incorrect to state that JavaScript code reasons about types. However, in order for JavaScript code to correctly evaluate, values must contain tags indicating whether they are strings, object, etc. Expressions such as primitive operators implicitly access these tags to function properly. JavaScript also provides operators that yield tag information directly: the `typeof`, `instanceof`, and `===` operators, and `for..in` loops.

JSTrace uses these operators to inspect values. Figure 3.2 represents the kinds of tags that JSTrace keeps track of. Some tags, such as $flat$, can be discovered directly with a primitive operator. Others, like $func$, can only be discovered by deeper inspection, which is accomplished by wrapping functions as explained in the next section.

When JSTrace provides output, it converts these tags into their closest representations in Typed JavaScript's type system. Most of these are straightforward. For example, flat value tags become the corresponding flat value types. Constructed object types, i.e. `InstanceOf<`*id*`>` tags, become nominal types taking the name of the constructor. One interesting case, however, is `Unknown`, generated when we have no information about a parameter or return value of a function. We could choose to simply output an error message, forcing the programmer to change that annotation before even attempting to type check. A more practical solution, however, is to output `Any`, signifying that the function can take or return a value of any type. This annotation is often incorrect. However, it sometimes passes type checking, for example in the case of an event handler that does not return a value. In these cases, it saves programmer time and effort, which is the goal of JSTrace.

3.3 Wrapping User Functions

The JSTrace compiler wraps all function statements and expressions with a call to a JSTrace function, `wrapFunc`, which returns a wrapped version of the function.

First, `wrapFunc` creates a unique tag name and adds it as a property of the function. A tag object is created and stored in a global map, indexed by this tag name. It is initialized with the number of named parameters, gotten with functions' `length` property, so even if a function is never called, JSTrace can at least provide a syntactically correct type annotation.

Whenever user code calls a user function, the wrapped version is called instead. It, in turn, inspects the values given and returned using a JSTrace function `inspectValue` (section 3.4), which

```
var foo = function() {
  if (this instanceof arguments.callee)
    alert("constr");
  else
    alert("func");
}
```

Figure 3.3: Attempt to check if a function was called as a constructor

returns tag objects for each value. The wrapped function then merges the new information with the information it already gathered. For example, if the first parameter in a given function was only known to be `int`, and a string was just observed, that parameter's tag becomes the union `int + string`.

3.3.1 Handling Constructors

Typed JavaScript provides nominal types for objects created by constructors. To infer nominal types, we must determine whether a function was called as a constructor. Unfortunately, it's impossible in JavaScript to tell how a function was called from inside the function. We might try checking if `this` was created by the current function we're in, which can be accessed by the special variable `arguments.callee`, as shown in figure 3.3. If `foo` is invoked as `foo()`, then `"func"` is alerted; if it is invoked as `new foo()`, then `"constr"` is alerted. However, this does not work in all cases:

```
var z = new foo();
z.foo = foo;
z.foo(); //alerts "constr", since "this" is bound to z, and z is instanceof foo
```

Instead of directly checking inside a wrapped function whether it was called as a constructor, JSTrace wraps `new` expressions with a call to `wrapNew`. This wrapper checks to see if the constructor being called is a tagged function. If it isn't, then it has been given a native constructor, e.g. Date, and it performs the `new` expression as JavaScript normally would. If it is given a tagged function then, before calling `new`, `wrapNew` marks the function to indicate that it is being called as a constructor. `wrapFunc` can then tell how a function is being called by checking to see if the function is marked.

If the function was called as a constructor, then the created object really has a nominal type. To indicate this, `wrapFunc` tags a constructed object with `InstanceOf<`*name*`>`, where *name* is the name of the constructor. If that object is ever inspected, this nominal tag is returned instead of a structural tag.

3.3.2 Handling "This"

`this` is valid in any JavaScript function. In many cases, however, functions never use `this`, even if they are part of an object:

```
var obj = {foo: function () { return 100; }};
obj.foo();
```

The type of `this` within the function is an object containing the function property `foo`. This is, in fact, a recursive type, and this type of recursion is not permitted by Typed JavaScript. However, the function in `foo` never uses `this`. Therefore, there is no need restrict the type of `this` in the function, and in this case, JSTrace will infer that `foo` can be called either as a regular function, or as a method of any object.

To account for this case, JSTrace wraps all calls to `this`. The wrapper is given a reference to the function it's in. The wrapper looks at the function's tag and updates it to indicate that the function does use `this`.

3.4 inspectValue

`inspectValue` turns values into tag objects.

Flat values First, `typeof` is used on the given value. If `typeof` returns `"number"`, `"string"`, `"boolean"`, or `"undefined"`, a flat value tag is returned. `"number"` is further distinguished into `"int"` and `"double"` by checking if the number is an integer.

```
inspectValue("hi") ↠ string
inspectValue(4) ↠ int
inspectValue(31.4) ↠ double
```

Constructed objects If `typeof` returns `"object"`, the object is checked to see if it has been constructed by a wrapped constructor. This indicates a nominal tag.

```
var MyClass = wrapFunc(function() {});
inspectValue(wrapNew(MyClass, [])) ↠ InstanceOf<MyClass>
```

Built-in objects If the object is not constructed, `inspectValue` uses `instanceof` to determine whether the object is one of the basic JavaScript object types, e.g. Date, or Number. Since these are basic JavaScript objects, there is no need to inspect them more deeply.

```
inspectValue(new Date()) ↠ Date
```

Arrays Arrays are built-in objects, but require more work. If the object is an array, the elements in the array are each inspected and their types are merged into a type union representing the type of the elements in the array.

```
inspectValue([1, 2, "string"]) ↠ Array<int + string>
```

Tagged objects Now we know we have a structural object. First, the object checked to see if it has been seen already. If it is, we return a reference to that object using its tag name, since we have seen the object already.

```
var obj = {a: 4};
inspectValue(obj) ⇝ {a : int}
inspectValue(obj) ⇝ Ref<obj>
```

Structural objects If the object was untagged, and therefore not seen before, then we assign the object a unique tag name. The object's properties are then examined using a `for..in` loop. `inspectValue` calls itself recursively on each property.

```
inspectValue({
    foo: "Bob", bar: {baz: 13.2}}) ⇝ {foo : string, bar : {baz : double}}
```

We must be careful to prevent infinite recursion, since an object could contain a reference to itself. However, any cycle must contain an object that has already been tagged. Once that object is inspected, a reference to its tag name will be returned, instead of it being deeply inspected. Therefore, an infinite cycle will not occur here. However, we must check for infinite recursion when converting the tags to types. When following references, we must check whether we have already seen the tag the reference points to. This indicates a type of recursion that Typed JavaScript does not support, so JSTrace outputs an error.

```
var obj = {};
obj.self = obj;
inspectValue(obj) ⇝ {self : Ref<obj>}
tagToType(inspectValue(obj)) ⇝ "{self : ERROR-RECURSION}"
```

Host objects The previous approach works on native JavaScript objects. However, JavaScript implementations can also implement "host" objects such as DOM elements in browsers or framework objects in Google Desktop [9, Section 4]. These objects can be troublesome. Some, such as the `framework` object in Google Desktop, don't allow addition of arbitrary properties, so they cannot be marked as being tagged. This can lead to infinite recursion. Others don't allow inspection with a `for..in` loop.

JSTrace checks whether inspection or tagging objects fails. After detecting a failure, there is nothing the analysis can do, so a `Host` tag is returned. On output, we could convert `Host` tags to the `Any` type, as with `Unknown`. However, knowing an object is a `Host` object can be an important hint to the programmer, so JSTrace requires the programmer to insert a manual annotation.

This isn't such a large defeat; usually, these objects are returned from API functions, and the API must be annotated anyway for type checking to function. It does limit the tool's usefulness in inferring API types, however.

```
inspectValue(framework) ⇝ Host
```

Tagged functions If `typeof` returns `"function"`, `inspectValue` checks to see if the function is tagged. If it is, it returns a reference to that function's tag:

```
var f = wrapFunc(function (a) { return a.toString(); });
inspectValue(f) ↠ Ref<f>
tagToType(inspectValue(f)) ↠ "(Any -> Any)"
f(4);
tagToType(inspectValue(f)) ↠ "(Int -> String)"
```

In this way, higher-order function types can be inferred:

```
var callF = wrapFunc(function (f, val) { return f(val); });
callF(f, 93);
inspectValue(callF) ↠ (Ref<f> * int -> string)
tagToType(inspectValue(callF)) ↠ "((Int -> String) * Int -> String)"
```

Since `callF` has a reference to `f`'s tag, information about `f` propagates to `callF`:

```
f("hey");
tagToType(inspectValue(callF)) ↠ "(((Int + String) -> String) * Int -> String)"
```

We must prevent infinite recursion here as well:

```
var a = wrapFunc(function(f) {return 5;});
var b = wrapFunc(function(f) {return "str";});
a(b);
b(a);
inspectValue(a) ↠ Ref<b> -> int
inspectValue(b) ↠ Ref<a> -> string
tagToType(inspectValue(a)) ↠ "(((ERROR-RECURSION -> Int) -> String) -> Int)"
```

Untraced functions If the function is not tagged, the function must have come from code that has not been wrapped. Since JavaScript provides no other tools to dynamically inspect functions, `inspectValue` gives up and returns an `"UntracedFunction"` tag to indicate this scenario. Once again, this could provide a useful hint to the programmer, so it is not converted to `Any` on output.

```
inspectValue(document.write) ↠ UntracedFunction
```

Typically, these are all the values that `typeof` can return. However, ECMA-262 indicates that `typeof` can return an implementation-specific value for other "host" objects [9, Section 11.4.3]. In this case, the tool has no recourse, so it treats the value as having a flat tag whose name is whatever string `typeof` returned. In practice, this has never happened.

3.5 Output

The simplest way to provide output is a list of annotations, one per line of output, where the first line is the annotation on the first function in the file, the second line on the second function, etc. To achieve this goal, JSTrace maintains a global array with each entry corresponding to each annotation. `wrapFunc` takes the function's position in the file as a parameter. When `wrapFunc` is called,

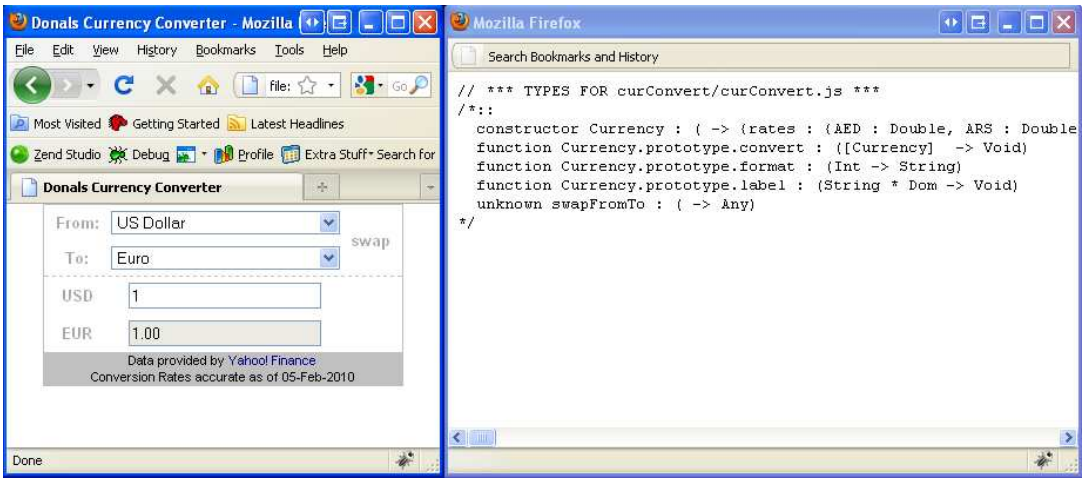

Figure 3.4: JSTrace output on Firefox

it sets the corresponding entry in the array to the function's tag. Thus, to produce the output, JSTrace simply iterates through the array, looking up each tag in the global map and converting it to a string representing the corresponding type.

There is a case where an entry in the array wouldn't be initialized. If a user function that creates nested functions is never called, then neither is the wrapper. To make sure that a syntactically correct function annotation is printed for each function, the compiler inserts a call to an initialization function, taking an array whose elements are the number of named arguments in each function. If a function is never even created, then JSTrace will consider its parameters and return value `Unknown`.

JSTrace, in principle, works on any implementation of JavaScript. However, providing output is implementation-dependent. Therefore, JSTrace currently only works on the Firefox and Google Desktop JavaScript environments.

Firefox Output on Firefox is provided via a pop-up window, which is updated every second.

Figure 3.4 is a screenshot for a traced version of a currency conversion tool. The swap button has never been pressed, so JSTrace doesn't yet know any information about `swapFromTo` except the number of arguments.

Google Desktop On Google Desktop, hooks are inserted into the right-click menu. Clicking on the proper menu option pops out a DetailsView containing the output, also updated every second (figure 3.5).

Using the Output There are two types of output: the human-readable one already shown, which aids with evaluating code coverage while running JSTrace, and a machine-readable one (figure 3.6).

JSTrace provides another program which takes the machine-readable version as input, along with the original JavaScript file, and stitches them together to provide an annotated version of the program. This annotated version can then be run through our type checker to verify type soundness.

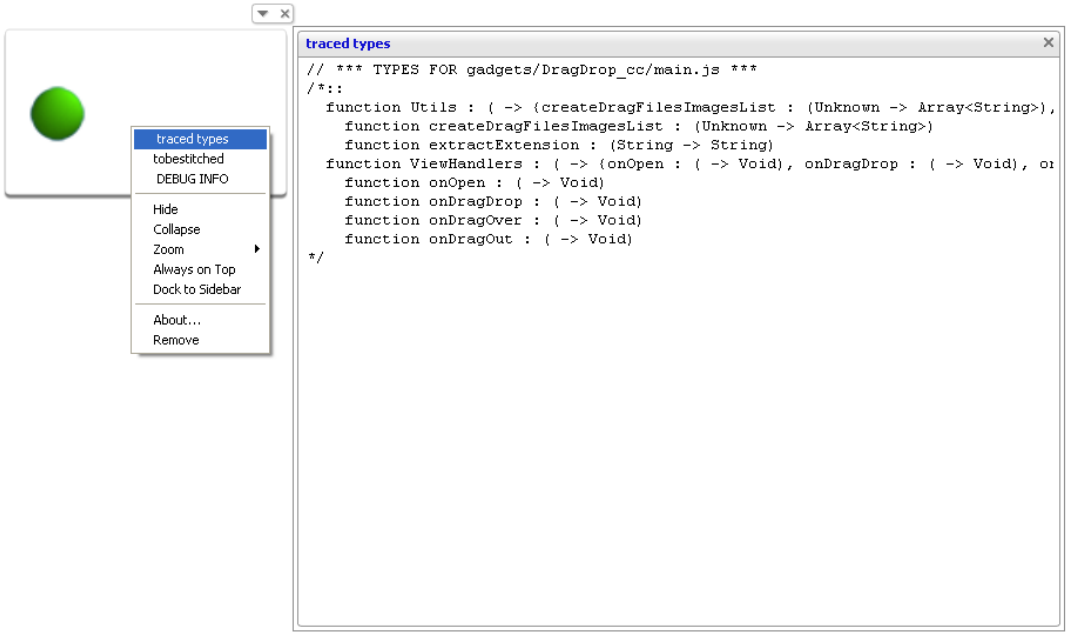

Figure 3.5: JSTrace output on Google Desktop

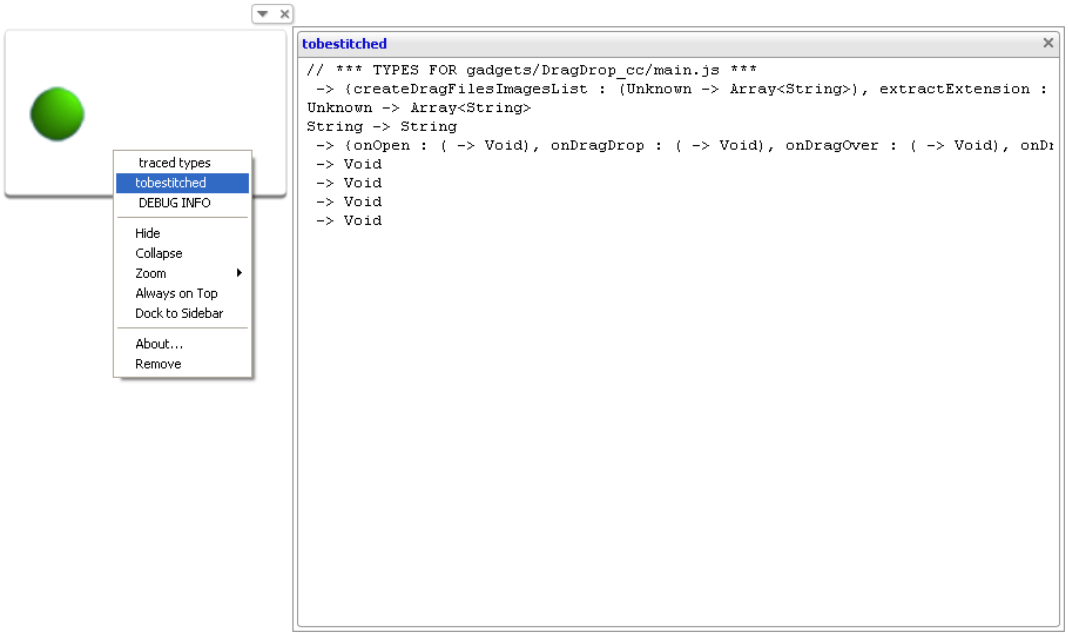

Figure 3.6: Machine output on Google Desktop

Chapter 4

Evaluation

I ran JSTrace on gadgets and browser programs. I ran the traced programs and used the discovered types to annotate the program. I then recorded how many more changes were necessary to get the program to type check.

In figure 4.1, "Changes" is the total number of changes necessary to get the program to type check. "Annotations" is how many of these changes were function annotations. "Refactorings" is the number of code changes unrelated to annotations that had to be made for the program to type check. This category of changes is mostly due mostly to idioms Typed JavaScript doesn't support, such as using uninitialized global variables, or passing eval strings to `setTimeout` instead of functions. The "Automated" column is the number of these annotations that the tool correctly created. The remainder required manual changes. The "Automated" column is shown both as a percent of total annotations required as well as total changes required.

As the table shows, JSTrace greatly lowered the number of annotations that had to be performed manually. For some programs, like `topten.js`, most of the burden lay not in the function annotations, but in correcting code that used certain idioms. In these cases, the impact of the tool was small, reducing the number of changes required by around 20%. However, in some, such as `text2wav.js`, the tool was able to do most of the work required to type the program, reducing the burden by around 75%. On average, 91% of annotations were successfully discovered by JSTrace, which was 46% of all changes that were required by Typed JavaScript.

The cost of running the traced programs themselves did not seem high. Creating a traced version of a program was straightforward and did not take long. The runs I performed usually required around a minute or two of clicking through all the possible user interface components. This was a small effort compared to reading and understanding most of the code before being able to begin annotating the functions. Since the tool cut the number of changes required in half without much effort, it is an effective way to reduce the burden of typing a program.

Filename	LOC	Changes (% of LOC)	Annotations (% total)	Refactorings (% total)	Inferred (% annots)	Inferred (% total)
watchimer.js	947	34 (3.59%)	17 (50.00%)	17 (50.00%)	15 (88.24%)	44.12%
countdown.js	129	8 (6.20%)	4 (50.00%)	4 (50.00%)	4 (100.00%)	50.00%
animation.js	70	5 (7.14%)	5 (100.00%)	0 (0.00%)	4 (80.00%)	80.00%
resistor.js	591	52 (8.80%)	32 (61.54%)	20 (38.46%)	32 (100.00%)	61.54%
morse.js	275	25 (9.09%)	12 (48.00%)	13 (52.00%)	12 (100.00%)	48.00%
text2wav.js	488	50 (10.25%)	41 (82.00%)	9 (18.00%)	38 (92.68%)	76.00%
hashapass.js	257	30 (11.67%)	20 (66.67%)	10 (33.33%)	14 (70.00%)	46.67%
rsi.js	328	49 (14.94%)	22 (44.90%)	27 (55.10%)	22 (100.00%)	44.90%
metronome.js	106	16 (15.09%)	12 (75.00%)	4 (25.00%)	10 (83.33%)	62.50%
catchit.js	165	25 (15.15%)	9 (36.00%)	16 (64.00%)	6 (66.67%)	24.00%
topten.js	443	85 (19.19%)	18 (21.18%)	67 (78.82%)	18 (100.00%)	21.18%
TOTAL	3799	379 (9.98%)	192 (50.66%)	187 (49.34%)	175 (91.15%)	46.17%

Figure 4.1: Changes required to type check programs

Chapter 5

Conclusion

I have created JSTrace, a tool that automatically annotates JavaScript programs with information gathered while running them. I have described how the tool uses JavaScript's built-in operators to gather tag information about values, and then converts these tags to types that Typed JavaScript understands. Finally, I evaluated the tool on real-world JavaScript code, showing that it is effective at reducing the burden of typing programs. Facilitating the process of porting existing code makes Typed JavaScript a more practical tool, and therefore helps tackle some of the concerns that plague Web development today.

Future Work I am interested in whether JSTrace could be automated further to remove the need for user interaction when running a traced program. Perhaps a method can be developed to automatically click and interact with GUI elements in a browser. Another possibility is to simulate a browser environment without the overhead of fully rendering Web pages and taking user input. It may also be possible to modify a browser's JavaScript interpreter to automatically trace all JavaScript code it executes, thus removing the need for a separate compilation step. This might even allow discovery of host objects, which cannot be deeply inspected with JavaScript alone. Discovering host object types could prove useful for typing APIs, which currently must be done manually.

I noticed that most of the function annotations JSTrace discovered did not exercise its full capabilities, such as discovering higher-order function types. I'm therefore also interested in how effective an unsound static inference algorithm would be in discovering correct annotations. Even though it would be difficult to get accurate results, Typed JavaScript function annotations might be simple enough to make even a rough approximation useful. The benefit would be to remove the overhead of compiling and running a program entirely.

Chapter 6

Related Work

Static Analysis Other forms of static analysis besides type systems have been proposed to deal with the problems JavaScript poses without replacing the language. One type of tool is a "lint" [16, 18, 19], which mostly detects code patterns that are considered bad form and can lead to bugs. These are of limited usefulness, as they only provide a superficial analysis of the code, and serve more as style guidelines.

Other tools provide a deeper analysis of the code [17, 12]. These fully-automated tools are often not practical, however. They yield imprecise results, and they do not scale, taking hours to run on larger programs. Web developers are used to immediately running their programs whenever they make modifications.

My approach is to use a type system. This yields accurate results that are more useful than those of a lint, and runs quickly even on large programs.

Semantics A type system requires a semantics to relate typing judgements to program execution. The ECMA specification [9] is informal and complex, consisting of 200 pages of prose and pseudocode, making it unsuitable to use. Maffeis, Mitchell, and Taly provide an operational semantics [20] based directly on the specification. It is still large at 30 pages, it omits a few syntactic forms, and inherits the standard's complexities. Other semantics only work over a small portion of JavaScript [15, 24].

For Typed JavaScript, my advisors and I are using a semantics we published called λ_{JS} [13]. λ_{JS} takes a different approach, presenting a simplified, conventional core semantics. The rest of JavaScript, with all its inherent complexity, is desugared to this core semantics. Desugaring together with the core is small, manageable, and covers all of JavaScript besides `eval`, making it suitable for use with a static type system tackling real world code.

Static Type Inference Static type inference has been attempted on JavaScript. Anderson et al. develop a semantics, a type system, and a sound static inference algorithm [2] for a subset of JavaScript. However, the subset excludes prototypes, first-class functions, nested functions, and global variables, rendering their system unusable for most real JavaScript programs.

B. Cannon presents a form of type inference for Python [5]. His goals are different than mine, however, aiming to improve the performance of Python code by using types to removing run-time type checks. He concludes that Python is not geared towards type inference, noting that Python's properties interfere too much with type inference for it to be effective. JavaScript shares many of these properties, indicating that static type inference might be an incorrect choice for JavaScript.

A. Aiken and B. Murphy present a type inference algorithm for a dynamically typed language, FL [1]. Their goal was also to improve performance. FL is significantly different from JavaScript and lacks many of its thorns. They also cite performance issues with their algorithm, stating that it is exponential in the worst case.

Michael Furr et. al present Diamondback Ruby [11], a static type system along with an inference algorithm for the dynamic language Ruby. Like Typed JavaScript, their type system provides union types. However, their type checker provides no way of discriminating union types with control operators. Their type system is unsound, generating warnings instead of failing on code that is potentially unsafe. Typed JavaScript's goal is to prevent potentially unsafe code from being published on the Web, and therefore aims to statically reject programs containing errors. Furthermore they, too, cite performance problems on larger code bases.

R. Cartwright and M. Fagan present Soft Typing [6], a system for bringing type checking to dynamic languages. The goals of soft typing are to pass as many programs as possible and to require no annotations from the programmer. Programs that the system cannot prove correct are wrapped to instead fail at run-time. As before, we wish to reject programs before they are published ont he Web.

Mixing static and dynamic code Gradual Typing [23] attempts to migrate code to use static types. It offers a representation for dynamic types in a static type system, allowing static and dynamic code to mix freely while still offering some guarantees about statically known parts of a program. While one of Typed JavaScript's goals is to support migration to a static type system, we would do so only with clean separation between the typed and untyped parts of a program by using contracts [10]. It is also unclear whether Gradual Types work on real world programs, whereas we have evaluated Typed JavaScript on actual code and shown it to be effective.

Bibliography

[1] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[3] 'Most websites' failing disabled, 2006. http://news.bbc.co.uk/2/hi/technology/6210068.stm [accessed 03-May-2010].

[4] Google searches web's dark side, 2007. http://news.bbc.co.uk/2/hi/technology/6645895.stm [accessed 03-May-2010].

[5] Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

[6] Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.

[7] Symantec Corporation. Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary), 2008. [Online; accessed 03-May-2010].

[8] Symantec Corporation. Symantec Internet Security Threat Report: Trends for 2009, 2010. [Online; accessed 03-May-2010].

[9] ECMAScript language specification, 3rd edition, 1999.

[10] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices Volume 37, Issue 9 (September 2002)*, 2002.

[11] Michael Furr, Jong hoon An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *2009 ACM Symposium on Applied Computing*, 2009.

[12] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *International World Wide Web Conference*, 2009.

[13] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[14] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. A practical type system for JavaScript. In *In Submission*, 2010.

[15] Dave Herman. Classicjavascript. http://www.ccs.neu.edu/home/dherman/javascript/ [accessed 06-May-2010].

[16] Javascript lint. http://www.javascriptlint.com/ [accessed 05-May-2010].

[17] Simon Holm Jensen, Anders Moller, and Peter Thiemann. Type analysis for javascript. In *The 16th International Static Analysis Symposium*, 2009.

[18] Jslint: The javascript code quality tool, 2010. http://www.jslint.com [accessed 05-May-2010].

[19] Jsure: The javascript checker, 2008. http://www.jsure.org/about [accessed 05-May-2010].

[20] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[21] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, 1978.

[22] Jens Palsberg. Type inference for objects. In *Computing Surveys*, 1996.

[23] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

[24] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.