Type Safety for JavaScript

Ben Weber
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064
bweber@ucsc.edu

Abstract

JavaScript is a client-side scripting language for the web and has become increasingly popular with the introduction of AJAX. JavaScript is a dynamic-typed language and is closer to a functional language than a procedural language. Runtime errors can occur in JavaScript due to invalid type conversions and accesses to undefined members of objects. One solution is the use of type systems to validate script before it is executed. Current work has focused on implementing type inference in JavaScript to statically check for errors. Additionally, the development of the JavaScript 2.0 specification will enhance the type safety of JavaScript applications.

1 Introduction

JavaScript is a scripting language for the web, originally named ECMAScript [6]. It is a dynamic, weakly typed language with first-class functions. JavaScript is weakly typed in the sense that is has objects, but no classes. It relies on prototyping instead of inheritance to share and extend functionality in the tradition of the Self language [16]. The prototyping functionality of JavaScript makes it an expressive scripting language, but also exposes an environment in which runtime errors can occur.

JavaScript is a flexible language that allows dynamic addition of members to objects. Due to this functionality, static type checking of code is not performed. Lack of type safety makes the development of bug-free applications a difficult task. Runtime errors can occur if a script attempts to access a member of a class that has not been initialized or invokes a function that does not exist. Taken together with the possible differences in the object hierarchies, it can become very hard to debug and maintain JavaScript programs [15]. Additionally, debugging tools for JavaScript have only recently become available [5].

The lack of type safety in JavaScript presents security vulnerabilities. Type errors may result in a buffer overflow [3], causing a web browser to behave erratically. Malicious errors may enable access to private information. Traditionally, a sandbox policy is implemented to ensure browser safety. However, the problem with the current solution is that scripts may conform to the browser policy, but violate the security of the system [10]. Since certain security vulnerabilities arise from the lack of type safety, enforcing a type system may prevent these vulnerabilities.

There are four main approaches for implementing type safety in JavaScript. The first approach adds a static type system to JavaScript [2, 15]. This method enables static checking of code, but requires type annotations and limits the flexibility of JavaScript programs. The second approach uses type inference [1] to reduce the amount of

annotation required, while ensuring type safety. Another approach is to verify type safety of a JavaScript program using a type checker implemented in ML [11]. The fourth approach uses instrumentation of JavaScript code to ensure type safety [17].

There are several tradeoffs to consider when selecting a technique for implementing type safety in JavaScript. If a static type system is used, then additional annotations are required. Also, enforcing static types limits backwards compatibility with earlier versions of JavaScript. If a dynamic type system is used, programmers can make use of flexible types. However, ensuring type safety for dynamic type systems is difficult. Additionally, browser implementations for type inference may not be equivalent. In order to support backwards compatibility while providing flexibility and satisfactory performance, JavaScript needs to support interoperation between static and dynamic typing. This style of system has become popularly known as gradual typing [14].

One of the main motivations for implementing type safety in JavaScript is to provide development tools for assisting well-intended developers to make bug-free applications. Current development tools, such as JSLint [4], are limited to syntactical checking of JavaScript code. Only limited support is available for IDE features such as auto-completion, because JavaScript allows the addition of new members at runtime. Implementing type inference in JavaScript would make it possible to determine the fields of an object at compile time, enabling the development of useful tools for building JavaScript applications.

This paper proceeds as follows. Section 2 provides an overview of the JavaScript programming language. Section 3 explores vulnerabilities in JavaScript. Static checking of JavaScript applications is discussed in Section 4 and type systems for JavaScript are discussed in Section 5. Section 6 gives guidelines for web toolkits. Section 7 discusses directions for future work.

2 Programming in JavaScript

JavaScript is a classless, object-based language with first-class functions. In addition to the usual support for imperative programming, JavaScript has lexically scoped first-class functions so it may be called a functional programming language [15]. The primary data type of JavaScript is the object, which is a table of properties for fast lookup and update.

```
> var person = { name: "Bob", toString: function() { return this.name }}
> person.toString()
  Bob
> person.boss = "Alice"
  Alice
> person.setBoss = function(b) { this.boss = b}
  function (b) { this.boss = b; }
> person.setBoss("John")
> person.boss
  John
```

Figure 1: Classless JavaScript Example

A script demonstrating the classless and dynamic properties of JavaScript is shown in Figure 1. The script creates a person object with a field specifying the person's name and a toString function that returns the name. After instantiating the object and printing the name, a new field is added to the person specifying the person's boss. Next, a mutator for the boss field is added to the object. The mutator is then invoked, setting the boss field to a new value. The script demonstrates two features that make type checking in JavaScript difficult: user-defined objects can be created without the definition of a class, and member variables and functions can be dynamically added to objects.

Functions in JavaScript do not require type annotations for input parameters or return types. Parameters are passed to a function as a single array accessible with the arguments keyword. An example script showing how to define and invoke a function is shown in Figure 2. The script initially defines an addition function which takes the input parameters a and b. The addition function is then redefined using the arguments array, resulting in an equivalent function. Next, the function is called with parameters consisting of numbers, strings, too few arguments and too many arguments. The script demonstrates that functions in JavaScript will accept any number of arguments and do not enforce input or output types.

```
> function add(a, b) { return a + b }
> function add() { return arguments[0] + arguments[1] }
> add(1, 0)
  1
> add("Hello ", "World")
  Hello World
> add()
  NaN
> add(1,2,0)
  3
```

Figure 2: Functions in JavaScript

JavaScript is a loosely typed programming language. Variables are defined using the var keyword, which places no restrictions on the type of a variable. A variable can be defined as an integer and later used as a string or boolean type. The following types are provided in JavaScript: number, boolean, string, object and function. The type of a variable can be determined using the typeof keyword, demonstrated in Figure 3. JavaScript uses the inferred type of variables to determine the operation to perform. For example, it is necessary to determine whether the "+" symbol is intended for addition of numbers or concatenation of strings.

JavaScript allows a variable to modify its type based on the context of its usage. A script demonstrating the ability of a variable to change types is shown in Figure 4. The variable x is initially defined as a number. Applying the NOT operator twice converts the variable to a boolean type. Next, the variable is cast to a string and concatenated with another string. Applying the NOT operator twice casts the string to a boolean and

performing an addition casts the boolean to an integer. Finally, two strings are concatenated and then converted to an integer to perform subtraction.

```
> typeof 0
  number
> typeof true
  boolean
> typeof "string"
  string
> typeof (new Array())
  object
> typeof function f() {}
  function
```

Figure 3: JavaScript Types

```
> var x = 1
> x = !!x
  true
> x += "string"
  truestring
> x = 10 + !!x
  11
> x = "40" + "1" - x
  390
```

Figure 4: Type Conversions in JavaScript

3 Vulnerabilities in JavaScript

The ability to dynamically add and update members in JavaScript results in potential runtime errors. There are several types of runtime errors in JavaScript: applying an arithmetic operator to an object that is not a wrapper object for a number, accessing a non-existent variable, and accessing a property of a null object. The cause of many runtime errors in JavaScript is due to invalid type conversions and the lack of classes.

```
> "hello " - "world"
  NaN
> "5" / "one"
  NaN
> "true" & false
  0
> "true" - false
  NaN
```

Figure 5: Arithmetic Errors in JavaScript

Type conversion errors occur in JavaScript when arithmetic operations are performed on non-numeric values. If operands cannot be cast to numeric types, a runtime error is thrown. Examples of runtime type-conversion errors are show in Figure 5. The first operation attempts to subtract two strings, resulting in an invalid number. The second operation attempts to divide two strings representing numbers, but is invalid because the second string is non-numeric. The third operation demonstrates the unpredictability of type conversions, since boolean values are provided and an integer type is returned. The last operation shows another example of unexpected behavior, because the expected result of 1 is not returned. Many of these errors could be flagged before execution using a type-checking tool.

Runtime errors can occur in JavaScript due to the lack of enforcement of function parameters. The number of formal parameters in a function definition need not match the number of actual parameters. If there are too few parameters, the remaining parameters are taken to be undefined [15]. An example of a runtime error due to invalid parameters is shown in Figure 2. When the add function is called without passing input parameters, the addition operation fails. Since the specification of function parameters is optional in JavaScript, runtime errors can occur due to undefined variables or invalid types.

3.1 Prototype-Based Errors

JavaScript is a prototype-based language that supports the dynamic addition of members to objects. Runtime errors occur when a script references an undefined variable or non-existent member of an object. A prototype-based error in JavaScript is shown in Figure 6. An object is created with a single member variable. The value of the member variable is then printed. Then the script attempts to access the member variable y, which is undefined. Next, the script modifies the value of undefined, which is used incorrectly in the last line.

```
> var obj = { x:1 }
> print(obj.x)
  1
> print(obj.y)
  undefined
> obj["undefined"] = "should be undefined"
> print(obj[obj.y])
  should be undefined
```

Figure 6: Prototype Errors in JavaScript

3.2 Classes in JavaScript

JavaScript is a classless programming language. Despite this property, one of the most common idioms found in contemporary JavaScript applications is the emulation of class-based object orientation through the prototype system [11]. Classes are useful constructs for implementing large applications in JavaScript. However, all members of an object are public. Therefore, prototype-based errors can still occur when using classes. Future versions of JavaScript will solve this problem by standardizing a system of classes and interfaces similar to Java or C# [9, 7].

4 Static Analysis of JavaScript Code

Static code-checking tools provide a mechanism for detecting specific runtime errors in JavaScript. Static code checkers detect syntactical errors, such as undefined global variables. However, syntax-based tools are unable to determine at compile time if member references are valid. Another approach to verify JavaScript applications at compile time is static type systems. If a program passes the type checker, then certain runtime errors do not occur. Static type systems are further explored in Section 5.

JSLint is a JavaScript syntax checker and validator [4]. It provides warnings for the following errors: undefined global variables, undefined functions, invalid semicolon usage, invalid block structure, unreachable code and poor coding style. Since JavaScript is a loosely typed, dynamic-object language, it is not possible to determine at compile time if member variables are defined before usage. However, JSLint provides assistance to help detect erroneous references to member variables. It generates a summary containing all of the values that were used with dot notation to name the members of objects. The intended use of this summary is to find misspelled member names. This functionality also provides a mechanism for tracking unsafe type operations.

Code checking tools enable detection of specific runtime errors, but are targeted at developers, not users. It is possible for a script with poor coding style to be type safe, but fail a syntax checker. Therefore, to validate type safety at compile time, several type systems have been proposed for JavaScript.

5 Type Systems in JavaScript

There are two main approaches for implementing type safety in JavaScript. The first approach is to implement a static type system to flag suspicious operations at compile time. Static type systems may require type declarations or only work for a subset of the JavaScript language. An alternate approach is to use a type inference algorithm to determine unsafe operations.

5.1 Static Typing

In a statically typed language, the data type of every variable, parameter and function return value is known at compile time. Type information for data types is specified through declarations. A type specifies the possible states of an object. If an additional member is added to an object, new states are possible. Therefore, adding or modifying the type of a member of an object changes the type of the object. A static type system for JavaScript needs to track the assignment of members to objects to track the types in a program.

Anderson and Giannini [2] present a static type system for JavaScript that detects accesses to non-existent members at compile time. The goal of the system is to maintain the flexible programming style offered by JavaScript and provide the safety offered by a static type system. Anderson and Giannini demonstrate the type system with a subset of JavaScript, JS_O. Members of objects in JS_O are defined as definite or potential, where potential members become definite upon assignment. The system enables specification of static types in a dynamically typed language.

```
> function Person(x) {
>   this.money = x;
> }
> function employPerson(x, y) {
>   x.boss = y;
>   x;
> }
> var john = new Person(100);
> var paul = new Person(0);
> employPerson(paul, john);
```

Figure 7: Untyped JS_O Person Example

In JS_O, type annotations are added to JavaScript code in order to specify typing information for input parameters as well as return values. An example of an untyped JS_O script is shown in Figure 7. In the script, two person objects are instantiated using a constructor. The second person is then set as the boss of the first person using the employPerson function.

```
> function Person(x:Int):t1 {
>       this.money = x;
> }
> function employPerson(x:t1, y:t1):t2 {
>       x.boss = y;
>       x;
> }
> t1 john = new Person(100);
> t1 paul = new Person(0);
> employPerson(paul, john);
```

Figure 8: Typed JS_O Person Example

A typed version of the script is shown in Figure 8. The return types t_1 and t_2 have been added to the constructor and employPerson function. Also, type annotations have been added for function parameters as well as variable declarations. The types t_1 and t_2 are both person types, but contain different definite members. The function employPerson returns type t_2, because the boss member becomes definite in the body of the function. Invoking the employPerson function causes the type of Paul to change from t_1 to t_2.

JS_O prevents runtime errors caused by accessing non-existent members of objects and the system is sound with respect to the operational semantics given. However, JS_O does not support dynamic variable creation, functions as objects, dynamic removal of members, delegation and prototyping, because this is too difficult to support in a dynamically typed language [2]. The main drawback of JS_O is the amount of annotation required to specify type declarations.

Other work on static type systems has focused on type conversions [15]. Thiemann defines a type system that tracks the possible traits of an object and flags suspicious conversions. The system is guided by a matching relation, which specifies type convertibility. If a script violates a matching relation, it is flagged or rejected based on the severity of the error. The type system covers a representative subset of JavaScript and does not require type declarations. However, the system is unable to detect accesses to non-existent members at compile time.

5.2 Type Inference in JavaScript

Type inference systems allow programmers to benefit from the safety offered by a static type system, while leaving programmers free to omit type annotations. Implementing type inference for JavaScript is difficult, because it is a weakly typed programming language. Type inference works by reducing expressions to implicitly typed values, which are used in place of type annotations. However, it is not always possible to determine the implicit type of an object, requiring the use of explicit type annotations.

Anderson et al. [1] present a technique for implementing type inference. A non-typed program is converted into a typed program. If the conversion succeeds, a static type checker [2] verifies the type safety of the program. Non-typed programs are typed using a constraint satisfaction algorithm. The algorithm generates a set of type variables with constraints between them, based on the usage of definite and potential members of objects. The algorithm attempts to find a solution that satisfies all of the type constraints. If a solution is found, the program is annotated with type declarations and verified for type safety. If no solution is found, the type safety of the program cannot be determined.

Type inference can be applied to JavaScript by converting non-typed programs into typed programs. However, determining if a program is typeable is quite difficult to achieve for recursive type systems [1]. It may be necessary for some type annotations to be given by the user.

Hirotaka et al. [12] present a type inference system for JavaScript using theorem provers based on model generation. They propose a type checking system for a subset of JavaScript and construct a simple type inference system for this subset. Hirotaka et al. [13] extend this work by demonstrating that a formal approach to type inference can be applicable to practical programming languages. However, the systems presented by Hirotaka et al. are limited to subsets of JavaScript.

5.3 Instrumentation of JavaScript Code

JavaScript has the ability for script to generate additional script, known as higher-order script. Therefore, static checking of code may be insufficient to ensure type safety. All of the type systems examined so far have been unable to ensure type safety for higher-order script. Yu et al. [17] propose runtime instrumentation of JavaScript code to enforce security policies for higher-order script. JavaScript code goes through a rewriting process and callbacks are embedded in the instrumented code, generating script that can be carried out on demand. Instrumentation of JavaScript code provides a mechanism for ensuring type safety of higher-order script.

5.4 JavaScript 2.0

The Ecma TC39-TG1 working group is using ML as the specification language for the next generation of JavaScript [11]. JavaScript 2.0 will support Class-based OOP, name management and gradual typing. The name management features will enable private fields, preventing clients from guessing object properties that are meant to be internal. The use of a gradual type system introduces type declarations while maintaining compatibility with earlier versions of JavaScript. Herman and Flanagan use ML to concisely formalize type soundness for JavaScript: if a JavaScript program passes the type checker, then certain run-time errors do not occur.

6 JavaScript Toolkits

Several toolkits have been developed to assist programmers in the development of JavaScript applications, such as the Dojo toolkit [8]. Web toolkits provide standardized interfaces for performing common tasks on several browsers. Web frameworks ease development, because applications work across different versions of JavaScript. Toolkits also improve application security, because they are stable, proven code bases.

Web toolkits provide standard interfaces for performing common tasks. However, JavaScript does not enforce correct usage of interfaces, since a script may supply invalid parameters or too few parameters to a method. Therefore, static type checking should be applied to toolkit interfaces to prevent incorrect usage. Type checking can be achieved by adding type declarations to toolkit interfaces and verifying that the input parameters meet the annotated types. Toolkits could use a system of gradual typing to provide static types for interfaces and utilize dynamic types to implement functionality. Web frameworks can encapsulate type annotations, easing the transition from the current version of JavaScript to a type safe version of JavaScript.

7 Conclusion

The lack of type safety in JavaScript presents security vulnerabilities. This paper has explored several approaches to improve type safety in JavaScript applications. Static type systems offer the greatest safety, but are the most restrictive. Type inference systems provide the flexibility of a dynamic type system, while providing the safety of a static system. However, practical implementation of the full language has been shown to be a difficult task. Other approaches such as instrumentation of JavaScript code provide practical mechanisms for improving type safety. The most promising area of work is the development of a new JavaScript specification, which will enable proper classes and information hiding.

Currently, type systems are a useful tool for development of JavaScript applications. Type systems could be used to build powerful integrated development environments, providing compile-time checking and debugging tools. However, the implementation of type inference systems or runtime instrumentation may still be impractical for web browsers. Therefore, web browsers should be responsible for catching and handling type errors in JavaScript. In order to limit barriers to adoption, JavaScript 2.0 must carefully weigh the tradeoffs between type safety, implementation complexity, backwards compatibility and performance.

References

1. C. Anderson, F. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. Ecoop 2005-Object Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005: Proceedings, 2005.

2. C. Anderson and P. Giannini. Type Checking for JavaScript. Electronic Notes in Theoretical Computer Science, 138(2):37-58, 2005.

3. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems], pages 227-237, 2003.

4. D. Crockford. The Java Script Verifier, 2002.

5. D.M. Doolin, N. Sitar, and S. Glaser. The Firebug Project. Homepage: http://firebug. sourceforge.net/, Stand Nov, 2004.

6. E.S. ECMA. 262. ECMAScript Language Specification, 3(5):47, 1999.

7. E.S. ECMA. 334: C# Language Specification. ECMA, December, 2001.

8. Dojo Foundation. Dojo, the JavaScript toolkit.

9. J. Gosling. The Java Language Specification. Addison-Wesley Professional, 2000.

10. O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), June, 2005.

11. D. Herman and C. Flanagan. Status report: specifying javascript with ML. Proceedings of the 2007 workshop on Workshop on ML, pages 47-52, 2007.

12. O. Hirotaka, Y. Shin'Ichiro, S. Toshiki, and I. Yasuyoshi. A Formal Approach to Type Inference System for JavaScript Programs. IEIC Technical Report (Institute of Electronics, Information and Communication Engineers), 104(47):13-18, 2004.

13. O. Hiortaka, Y. Shin'Ichiro, S. Toshiki, and I. Yasuyoshi. A JavaScript Type Checker based on Model Generation Theorem Prover. IEIC Technical Report (Institute of Electronics, Information and Communication Engineers), 105(25):25-30, 2005.

14. J.G. Siek and W. Taha. Gradual typing for functional languages. Scheme and Functional Programming Workshop, September, 2006.

15. P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005: Proceedings, 2005.

16. D. Ungar and R.B. Smith. SELF: The power of simplicity. Higher-Order and Symbolic Computation, 4(3):187-205, 1991.

17. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 237-249, 2007.