### 英文

The JavaScript programming language is widely used for web programming and, increasingly, for general purpose computing.As such, improving the correctness, security and performance of JavaScript applications has been the driving force for research in

type systems, static analysis and compiler techniques for this language. Many of these techniques aim to reign in some of the most dynamic features of the language, yet little seems to be known about how programmers actually utilize the language or these features. In this paper we perform an empirical study of the dynamic behavior of a corpus of widely-used JavaScript programs, and analyze how and why the dynamic features are used. We report on the degree of dynamism that is exhibited by these JavaScript programs and compare that with assumptions commonly made in the literature and accepted industry benchmark suites.

Keywords：Dynamic Behavior, Execution Tracing, Dynamic Metrics, Program Analysis, JavaScript

1. Introduction

JavaScript is an object-oriented language designed in 1995 by Brendan Eich at Netscape to allow non-programmers to extend web sites with client-side executable code. Unlike more traditional languages such as Java, C# or even Smalltalk, it does not have

classes, and does not encourage encapsulation or even structured programming. Instead JavaScript strives to maximize flexibility. JavaScript’s success is undeniable. As a data point, it is used by 97 out of the web’s 100 most popular sites.2 The

language is also becoming a general purpose computing platform with office applications, browsers and development environments being developed in JavaScript. It has been dubbed the “assembly language” of the Internet and is targeted by code generators from the likes

of Java and Scheme. In response to this success, JavaScript has started to garner academic attention and respect. Researchers have focused on three main problems: security, correctness and performance. Security is arguably JavaScript’s most pressing problem: a number of attacks have been discovered that exploit the language’s

dynamism (mostly the ability to access and modify shared objects and to inject code via eval). Researchers have proposed approaches that marry static analysis and runtime monitoring to prevent a subset of known attacks. Another strand of

research has tried to investigate how to provide better tools for developers for catching errors early. Being a weakly typed language with no type declarations and only run-time checking of calls and field accesses, it is natural to try to provide a static type system

for JavaScript. Finally, after many years of neglect, modern implementations of JavaScript have started to appear which use state of the art just-in-time compilation techniques.

2. Motivation and Related Work

The original impetus for our interest in JavaScript was to assess the feasibility of a static type system able to successfully and meaningfully type check existing JavaScript programs. Other dynamic languages such as Scheme have had recent success introducing

gradual typing systems, but they have stopped short of type checking object-oriented extensions (implemented by macros in Scheme). For JavaScript, Anderson et al. proposed a type system with definite and potential types, while Heidegger and Thiemann following up on some of their earlier workpropose recency types in, and Furr et al. proposed a related system for DRuby. While all of these type systems acknowledge some minor simplifications to the target language, they rely on fairly similar assumptions. For instance, Thiemann writes: “Usually, no further properties are defined after the initialization and the type of the properties rarely changes.”

Common Assumptions about the dynamic behavior of JavaScript.We proceed to enumerate the explicit and implicit assumptions that are commonly found in the literature and in implementations.

1. The prototype hierarchy is invariant. The assumption that the prototype hierarchy does not change after an object is created is so central to the type system work that chose to not even model prototypes. Research on static analysis typically does not mention prototype updates. Yet, any modification to the prototype hierarchy can potentially impact the control flow graph of the application just as well as the types of affected objects.

2. Properties are added at object initialization. Folklore holds that there is something akin to an “initialization phase” in dynamic languages where most of the dynamic activity occurs and after which the application is mostly static. For JavaScript this is embodied by the assumption that most changes to the fields and methods of objects occur at initialization, and thus that it is reasonable to assign an almost complete type to objects at creation, leaving a small number of properties as potential.

3. Properties are rarely deleted. Removal ofmethods or fields is difficult to ccommodate in a type system as it permits nonmonotonic evolution of types that breaks subtyping guarantees usually enforced in modern typed languages. If deletion is an exceptional occurrence (and one that can be predicted), one could use potential types for properties that may be deleted in the future. But, this would reduce the benefits of having a type.

system in the first place, which is probably why related work chooses to forbid it. Static analysis approaches are usually a bit more tolerant to imprecision and can handle deletes, but we have not found any explanation of its handling in existing data flow analysis techniques.

4. The use of eval is infrequent and does not affect semantics.The use of eval

on arbitrary strings has the potential of invalidating any results obtained by static analysis or static type checking. Thus many works simply ignore it, while other assume that uses are either trivial or related to deserialization using the JSON protocol .

5. Declared function signatures are indicative of types.Type systems for JavaScript typically assume that the declared arity of a function is representative of the way it will be invoked. This is not necessarily the case because JavaScript allows calls with different arities.

6. Program size is modest. Some papers justify very expensive analyses with the explicit assumption that handwritten JavaScript programs are small, and others implicitly rely on this as they present analyses which would not scale to large systems.

7. Call-site dynamism is low. Some JavaScript implementations such as Google V8 rely on well-known implementation techniques to optimize JavaScript programs such as creating classes (in the Java sense) for objects and inline caches. These techniques will lead to good performance only if the behavior of JavaScript is broadly similar to that of other object-oriented languages

8. Execution time is dominated by hot loops. Trace-based Justin-time compilers such as TraceMonkey rely on the traditional assumption that execution time is dominated by small loops.

9. Industry benchmarks are representative of JavaScript workloads. Standard enchmark suites such as SunSpider, Dromaeo and V8, are used to tune and compare JavaScript implementations and to evaluate the accuracy of static analysis techniques. But conclusions obtained from use of those benchmarks are only meaningful if they accurately represent the range of JavaScript workloads in the wild.

The goal of this paper is to provide supporting evidence to either confirm or invalidate these assumptions. We are not disputing the validity of previous research, as even if a couple of the above assumptions proved to be unfounded, previous work can still serve as a useful starting point for handling full JavaScript. But we do want to highlight limitations to widespread adoption of existing techniques and point to

**Related Work.** Until now, to the best of our knowledge, there has been no study of the dynamic behavior of JavaScript programs of comparable depth or breadth. Ratanaworabhan *et al.* have performed a similar study concurrently to our own, and its results are similar to ours [22]. There have been studies of Javascript’s dynamic behavior as it applies it applies to security [28][8], but the behaviors studied were restricted to those particularly relevant to securith. We conducted a small scale study of JavaScript and reported preliminary results in [19], and those results are consistent with the new results presented here. Holkner and Harland [14] have conducted a study of the use of dynamic features (addition and deletion of fields and methods ) ins the Python programming language. Their study focused on a smaller set of programs and concluded that there is a clear phase distinction. In their corpus dynamic features occur mostly in the initialization phase of programs and less so during the main computation. Our results sugges that JavaScript is more dynameic than Python isn practice. There are many studies of the runtime use of selected features of object-oriiented languages. For example, Garret *et al.* reported on the dynamisem of message sends in Self [11], Calder *et al.* characterized the difference of between C and C++ programs in [4], and Temporo *et al.* studied the usage of inheritance in Java in [23]. These previous papers study in great detail one particular aspect of each language. In this particular work, we strive for an overview of JavaScript, ande leave detailed analysis for future work. Finamlly, we were inspired by the work of Dufour *et al.* [7] and their rigorous framework for discussing runtime metrics for Java.

**3. Tracing and analysis Infrastructure**

The tracing infrastructure developed for this paper is based on an instrumented version of the WebKit web browser engine integrated into Apple’s Safari browser. While there are standalone interpreters available, they would not be able to deal with the mixture of DOM and AJAX that is commonplace in most JavaScript-enabled sites. For flexibility, analysis is performed offline. Our instrumented browser records a trace containing most operations performed by the interpreter ( reads, writes, deletes, calls, defines, etc.)as well as events for garbage collection and source file loads. Invocations to eval trigger anevent similar to the one for source file loads, and the evaluated string is saved and traced like any other part of the program’s execution, complete traces are compressed and stored to disk. While it does have some performance overhead, our instrumentation does not cause a noticeable slowdown in interactive applications, and none of our users complained about performance. Traces are analyzed offline and the results are stored in a database which is then mined for data. The offline tace analysis component is essentially an abstract interpreter for the event stream. It is able to replay any trace creating an abstract representation of the heap state of the corresponding JavaScript program. The trace analyzer maintains riche and customizable historical information about the program’s behavior, such a access histories of each object, call sites and allocation sites, and so on. Finally, several static analyses ( eval calsssification, code size metrics ) are performed on the recovered source files using the parsing framework from the Rhino JavaScript compiler.

As WebKit does not hide its identity to JavaScript code, it is possible for code to exhibit behavior peculiar to WebKit. Techniques like this are often used to work around bugs in JavaScript implementations or browsers. For instance, the Prototype JavaScript library includes the following check for WebKit

**4. Corpus and Methodology**

We have selected 100 web sites based on the Alexa list of most popular sites on the Internet,along with a number of sites of particular interest (including 280 slides, Lively Kernel, and a medley of different web sites visited in a single session). Morecover we also recorded traces for the three main industry benchmark suites (SunSpider, Dromaeo, and V8). For each of these sites we asked several of our colleagues to interact with the site in a “meaningful” manner Each interaction with a different web site was saved in adifferent trace. Multiple traces for the same site are averaged in our metrics.

In the remainder of this paper we focus on the results of 17 sites that we believe to be representative of the full range of behaviors and usage of popular libraries. The list of sites we have retained is shown in Figure 1. Data for all the web sites, as well as our tracing and analysis framework, database, and graphs are available on the project web site. For each site, we also list publicly-available JavaScript libraries utilized by the site, if any ,Sites that use the same libraries tend to have similar coding styles and program structure. It is instructive to see whether similarities also exist in the dynamic behavior of these programs, regardless of different application logic and use cases.

### 预览：

JavaScript编程语言是广泛用于web编程的语言，也越来越多地用于特定目的的计算。因此，如何提高正确性、安全性和JavaScript应用程序的性能就成为了推动类型系统分析，静态分析和编译技术研究等领域发展的动力之一。许多这些技术都旨在研究在一些最具有动态特性的语言，然而很少人会清楚的知道程序员是如何利用语言的这些特性。在本文中，我们将研究一些广泛使用JavaScript程序的动态行为，并分析如何以及为什么使用这些动态特性。我们将根据这些JavaScript程序得出具体的动态程度，并与目前参考文献中使用的假定数据和通用工业标准中的数据进行比较。

关键词：动态行为；执行跟踪；动态指标；程序分析；JavaScript

1、介绍

JavaScript是一种面向对象的语言，它可以通过客户端的可执行代码让用户实现和网站的交互，它是在1995年由Netscape公司的Brendan Eich设计出的。它不同于Java、C＃或Smalltalk这些传统语言，它不具有类，也不鼓励封装，甚至是结构化编程。与之对应的是，JavaScript十分强调编程语言的灵活性。JavaScript的成功是不可否认的。据相关数据显示，当前最流行的100个网站中有97个都使用了它.同时，JavaScript语言还成为了许多办公应用、浏览器应用以及开发环境等的通用计算平台。它一直被誉为互联网界的“汇编语言”，和Java以及Scheme语言类似，JavaScript需要代码生成工具的解释。由于这种成功，

JavaScript的已经赢得了学术界的重视和尊重，而研究人员也开始将目光都集中在三个主要问题：安全性，正确性和性能。安全性可以说是JavaScript所要解决的最紧迫的问题：到目前为止，已经发生了一些利用语言的动态性能进行恶意攻击的案例（主要是能够利用JavaScript访问和修改共享文件的特点，通过eval函数植入恶意代码）。研究人员已经提出一些解决方法，比如将静态分析和实时监控结合起来，以防御一些已知的攻击。另一个研究方向则致力于提供更好的检测工具，以帮助开发者尽早的发现程序中的漏洞。作为一种弱类型语言，JavaScript没有类型声明，并且只有再被访问和调用时才进行实时监控，因此为JavaScript提供静态类型的系统也就成为了很自然的事情。最后，在被忽视了很长时间以后，JavaScript实现很大的进步，目前最先进的即时编译技术得到了应用。

2、动机及相关工作

我们最初研究JavaScript的动力在于评估一个静态类型的系统是否能成功的验证现有的JavaScript程序。其他动态语言，如Scheme语言就成功的逐步应用到了类型系统中，但是其他方面的进展就相对有些停滞，比如缺乏有关面向对象的扩展的类型检查（Scheme中用宏来实现）。对于JavaScript，Anderson等人提出了一种具有明确类型和潜在类型的类型系统，而Heidegger和Thiemann等人则把他们之前的工作向前推进了一步，加入了一些最新的类型，Furr等人则为DRuby提出了一种相关的系统。所有这些类型系统都可以接受目标语言的一些简单的简化，但是他们要依靠相当类似假设。举例来说，Thiemann写道：“通常情况下，在初始化后几不会再定义其他的类型了，而且对象的类型也基本不会发生变化。”

常见的关于JavaScript的动态行为的假设。我们将在下文中继续列举一些

被公开发表或约定俗成的假设，这些假设已经在参考文献和具体程序中得到了广泛的应用。

1.prototype对象的层次结构是不变的。在创建一个新对象时，Prototype层次的结构不会改变的假设对于一个类型系统的运行是非常重要的，因此我们甚至不会选择给出prototype对象的模型。关于静态分析的研究很典型，通常它们不涉及到prototype的更新或变化，然而，任何关于prototype层次结构的改变都有可能影响到应用程序的控制流程，而那些受到影响的对象的类型也会出现变数。

2.在对象初始化时属性被添加。大家一般都认为，对于动态语言来说，存在一个类似于“初始化阶段”的时段，大部分的动态行为在这个时段发生，而在这个时段后应用程序大多都是静态的。那么具体到JavaScript的情形，就是假设作用域和对象方法的大部分变化都是在初始化阶段完成的，因此在创建对象时就已经确定类型的大部分，只留有一小部分属性待定这样的方法就显得合情合理了。

3.属性很少被删除。在一个类型系统中删除方法或作用域是很难的，因为这样一来就等于它允许了类型的发展非单调，这会破坏现代类型语言中默认执行的子类型化保证。如果删除的是一个特殊的情况（一个可以预测的情况），人们可以对这些未来可能被删除的属性使用一些潜在的类型。但是，这样做会减少我们使用类型系统所带来的好处，这可能是为什么一些相关的研究禁止这种做法的原因。静态分析方法通常更能接受这种不严密的做法，也可以对删除做出处理，但我们到目前还没有发现任何关于现有数据流分析技术的解释。

4.使用eval函数是罕见的而且不影响语义。Eval函数通常被用来计算任意字符串，它有可能使静态分析或静态类型检查得到结果是无效的。因此，很多文献就直接忽视了它，而另一些文献则假定这些使用的影响是微不足道的，或者使

用这些参考了JSON反序列化协议。

5.已声明函数的特征能够说明其类型。JavaScript 里的类型系统已经声明的函数的元数能够说明它在调用时所使用的方法，当然这种情况不是必须的，因为JavaScript允许不同元数的调用。

6.程序的大小是合适的。很多论文都花费大量笔墨明确指出，他们假定所研究的JavaScript程序是一些小型的程序，而且另一些则暗示了这一点，因为他们向大家介绍说现有的理论分析不能扩展到大型系统中去。

7.调用现场的动态行为很少。一些出色的JavaScript程序的完善和优化，如谷歌的V8 ，需要依赖一些已经成名的实现技术，比如对象创建类（就像Java一样）和内联缓存。如果JavaScript的行为大致类似于其它面向对象的语言，这些技术会带来良好的性能和使用效果。

8.执行时间主要由循环决定。一些基于跟踪的即时编译器，比如TraceMonkey等，它们工作的基础就建立在程序的执行时间主要由小循环决定这个传统的假设上。

9.行业基准能够反映JavaScript的工作负荷。如SunSpider，Dromaeo和V8这样的基准测试平台是用来评价和比较JavaScript的实现性能和计算静态分析技术的准确性的。但是，从使用这些基准测试平台得到的结论只有在一种情况下是有意义的，那就是它们能够准确地反映出在现实世界中运行的JavaScript程序的工作负荷区间。

本文的目的是提供支持或否定以上假设的证据。我们不是要质疑以前的研究成果的准确性，即使上述假设的一部分是被证明毫无依据的，以前的研究成果仍然可以为我们研究JavaScript提供一个有用的基础和出发点。但是我们想要强调

这些已经广泛采用的现有技术的限制，并且指出在今后的研究中要面临的挑战。

相关工作。到目前为止，据我们所知，还没有可以在深度和广度上具有相当规模的JavaScript程序的动态行为的研究。Ratanaworabhan等人已经进行过和我们类似的研究，其最终结果也是和我们相似的。以前在javascript应用到安全领域时也确实有一些关于其动态行为的研究，但是那些研究只侧重于和安全性紧密相关的部分。我们也曾进行过一些小规模的JavaScript研究，并得出了一些初步结果Harland，这些结果与这里提出的最新研究结果也是相吻合的。Holkner和也曾就Python语言动态特性的应用进行过研究（添加和删除作用域和方法），它们的研究重点集中在一系列小的程序上面，并得出了它们之间存在明确区别的结论。他们的研究表明，动态特性中的大部分都出现在程序的初始化阶段，而在程序的主要运算阶段，动态行为就逐渐减少了。我们的研究结果表明，在应用方面，JavaScript比Python更加动态化。关于面向对象语言在实时运行中的特性研究也有很多，比如Garret等人研究了Self语言中信息发送的动态程度，Calder等人则分析了C和C++程序运行中的差异，Temporo等人研究了Java中继承的应用等。之前的这些研究都对某种特定语言的某个特殊方面进行了非常详尽的研究，收到他们的鼓舞，我们决定争取为JavaScript也做出一个类似的概述，并为未来的研究留下一些详细的数据以供参考。最后，我们的工作得到了Dufour等人的启发，他们就Java的运行指标进行了严格的框架讨论。

3、跟踪和分析技术基础

本文对基础结构和跟踪的研究是基于WebKit的网页浏览器引擎的一个可以测量数据的版本，这个引擎现在被应用到了Apple的Safari浏览器上。虽然有可用的独立解释程序，他们却无法处理目前在大部分支持JavaScript的网站常见的

DOM和AJAX技术。为了提高实验的灵活性，我们的分析是在离线状态下进行的。我们的浏览器记录下由解释程序带来的大部分操作（读，写，删除，调用，定义等），同样作为垃圾收集和源文件加载的事件也会被跟踪。Eval函数触发一个特定事件的情形和源文件加载的情形是相似的，而经过计算的字符串也将被保存，并像程序执行时的其他部分一样被追踪分析。已经追踪到的痕迹被压缩并存储到磁盘中。虽然它在执行时有一些额外的开销，但是我们用来测试的工具在运行交互应用时的速度并没有受到影响，并且我们的用户也没有对性能有任何的抱怨。然后我们在离线状态下分析了这些数据，并将结果存储在一个数据库中进行数据挖掘。离线痕迹分析组件实质上是对抽象流的解释。它是能够重现这些痕迹的，并为相应的JavaScript程序得出一个关于堆状态的抽象解释。痕迹分析仪中把包括了关于程序动态行为的丰富的和可定制的历史信息，比如访问每个对象的历史，调用现场和分配网站等等。最后，一些静态分析（EVAL分类，代码大小的度量）是基于被恢复的源文件运行的，它使用了Rhino的JavaScript编译器的框架解析技术。

4、结论

本文提供了第一个大规模的研究JavaScript程序在运行时的行为。我们已经识别出一组字节代表实际的程序,这组字节的大小范围为数百千字节到兆字节，我们对每个网站都记录了很多痕迹,然后使用相应的编译仪器与离线分析工具从痕迹中提取了行为信息。我们使用此信息来评估清单中的9个关于JavaScript程序的常用假设。