Université de Montréal

A layered JavaScript virtual machine supporting dynamic instrumentation

par Erick Lavoie

Département d'informatique et de recherche opérationelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.) en informatique

Avril, 2013

Résumé

L'observation de l'exécution d'applications JavaScript est habituellement réalisée en instrumentant une machine virtuelle (MV) industrielle ou en effectuant une traduction source-à-source *ad hoc* et complexe. Ce mémoire présente une alternative basée sur la superposition de machines virtuelles. Notre approche consiste à faire une traduction source-à-source d'un programme pendant son exécution pour exposer ses opérations de bas niveau au travers d'un *modèle objet* flexible. Ces opérations de bas niveau peuvent ensuite être redéfinies pendant l'exécution pour pouvoir en faire l'observation. Pour limiter la pénalité en performance introduite, notre approche exploite les opérations rapides originales de la MV sous-jacente, lorsque cela est possible, et applique les techniques de compilation à-la-volée dans la MV superposée. Notre implémentation, Photon, est en moyenne 19% plus rapide qu'un interprète moderne, et entre 19× et 56× plus lente en moyenne que les compilateurs à-la-volée utilisés dans les navigateurs web populaires. Ce mémoire montre donc que la superposition de machines virtuelles est une technique alternative compétitive à la modification d'un interprète moderne pour JavaScript lorsqu'appliqué à l'observation à l'exécution des opérations sur les objets et des appels de fonction.

Mots-clés: Méta-circularité, Instrumentation, Dynamisme, Modèle Objet, Flexibilité, Performance, Machine Virtuelle, JavaScript

Abstract

Run-time monitoring of JavaScript applications is typically achieved by instrumenting a production virtual machine or through ad-hoc, complex source-to-source transformations. This dissertation presents an alternative based on *virtual machine layering*. Our approach performs a dynamic translation of the client program to expose low-level operations through a flexible *object model*. These low-level operations can then be redefined at run time to monitor the execution. In order to limit the incurred performance overhead, our approach leverages fast operations from the underlying host VM implementation whenever possible, and applies Just-In-Time compilation (JIT) techniques within the added virtual machine layer. Our implementation, Photon, is on average 19% faster than a state-of-the-art interpreter, and between 19× and 56× slower on average than the commercial JIT compilers found in popular web browsers. This dissertation therefore shows that *virtual machine layering* is a competitive alternative approach to the modification of a production JavaScript interpreter when applied to run-time monitoring of object operations and function calls.

Keywords: Metacircularity, Instrumentation, Dynamism, Object Model, Flexibility, Performance, Virtual Machine, JavaScript

Acronyms

Acronym	Definition
API	Application Programming Interface
JIT	Just-In-Time
JS	JavaScript
OO	Object-Oriented
OOPSLA	The International Conference on Object Oriented Programming, Systems, Languages and Applications
Pn	Photon Virtual Machine
Pn-fast	Photon Virtual Machine with a fast instrumentation
Pn-spl	Photon Virtual Machine with a simple instrumentation
SM	Mozilla SpiderMonkey Virtual Machine
V8	Google V8 Virtual Machine
VM	Virtual Machine

Remerciements

Un grand merci à mes colocataires des trois dernières années, Jérôme, Claire, Estelle puis plus tard Noël ainsi qu'à mes ami(e)s en particulier Awa et Bianka, qui ont été aux premières loges pour partager mes joies et frustrations mais surtout pour avoir offert un environnement qui m'a permis de décrocher quand j'en avais besoin.

Merci à Paul (Khuong) et Paul (Raymond-Robichaud) pour les discussions stimulantes, Étienne et David pour le partage de leur expérience du milieu académique, ce qui a brisé mon sentiment d'isolement aux moments où j'avais l'impression d'être seul face à certaines difficultés.

Merci également à tous ceux qui sont passés par le lab pendant mon séjour, en particulier Benoît, pour la co-création de la version la plus épique d'un Pacman 3D moustachu à laquelle j'aie participé, Eric, Vincent et Benjamin pour avoir démontré un intérêt pour mes nombreuses "démos" de versions intermédiaires du système ainsi que Maxime pour m'avoir montré un calibre de programmation que je n'avais pas connu jusqu'à mon entrée au lab.

Merci au personnel administratif du département d'informatique et de recherche opérationnelle, en particulier Mariette Paradis, pour les nombreux accomodements et les judicieux conseils prodigués pour la navigation au travers des requis administratifs inhérents à la poursuite d'études graduées dans un contexte institutionnel. Cela a grandement simplifié ma vie.

Merci à Bruno Dufour, qui aurait mérité d'être officiellement co-directeur vu son niveau d'implication dans mon projet de maîtrise. L'identification du problème d'instrumentation dynamique efficace est une conséquence directe de nos discussions. L'instrumentation dynamique a fourni un usage pratique à une solution qui était en recherche d'un problème. Tu es également le premier à avoir pleinement reconnu l'intérêt académique de mon travail.

Merci à Marc Feeley, mon directeur, d'avoir servi d'exemple spécifiquement par son soucis du détail et du mot juste ainsi que ses capacités techniques remarquables. Bien que ca n'ait pas toujours été intentionnel, merci également de m'avoir sorti de ma zone de confort. Mes limites personnelles ont été dans certains cas repoussées et dans d'autres cas raffermies.

Contents

Résumé ii

Abstract iii

Acronyms iv

Remerciements v

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Overview . . . 2

1.1.1 Design goals . . . 2

1.1.2 Overview of the Components . . . 3

1.1.3 Bottom-Up Overview . . . 4

1.2 Background material . . . 5

1.2.1 JavaScript . . . 5

1.2.2 Object model . . . 6

1.2.3 Function-calling protocol . . . 7

1.3 Outline . . . 7

2 Previous Work 8

2.1 Historical roots . . . 8

2.1.1 Openness . . . 8

2.1.2 Extensibility . . . 9

2.1.3 Dynamism . . . 10

2.1.4 Performance . . . 11

2.2 Instrumenting JavaScript VMs . . . 11

3 Design **15**

3.1 Message-sending foundation . . . 15

3.1.1 Semantics . . . 15

3.1.1.1 Message sending as a method call . . . 16

3.1.1.2 Reifying function calls . . . 16

3.1.1.3 Providing a memoization hook . . . 17

3.1.2 Translating reified operations to message-sending . . . 17

3.1.3 Efficient implementation . . . 18

3.1.3.1 Cache states and transitions . . . 20

3.2 Object representation . . . 23

3.2.1 Representation for objects . . . 23

3.2.2 Fixed arguments number specialization . . . 27

3.3 Compilation example . . . 27

4 Flexibility **30**

4.1 Obtaining object model operation information . . . 31

4.2 Obtaining a dynamic call graph . . . 33

4.2.1 Protecting the profiling code in a critical section . . . 35

4.3 Enforcing run-time invariants . . . 38

4.3.1 Ensure that all accesses are made to existing properties . . . 38

4.3.2 Ensure that a constructor always returns the object it receives . . . 39

5 Performance **41**

5.1 Setting . . . 42

5.2 Performance with no instrumentation . . . 43

5.3 Effect of send caching . . . 46

5.4 Comparison with interpreter instrumentation . . . 46

5.5 Performance with instrumentation . . . 48

6 Conclusion **50**

6.1 Limitations . . . 50

6.1.1 Accessing the `__proto__` property leaks the internal representation . . . 51

6.1.2 Meta-methods can conflict with regular methods if they have the same name . . . 51

6.1.3 Setting the `__proto__` property throws an exception . . . 51

6.1.4 Operations on *`null`* or *`undefined`* might throw a different exception . . . 51

6.1.5 Limited support for eval . . . 52

6.1.6 Function calls implicitly made by the standard library are not intercepted . . . 52

6.2 Future work 52
6.2.1 Improve compilation speed 53
6.2.2 Allow a finer-grained redefinition of meta-methods 53
6.2.3 Efficient instrumentation 53
6.2.3.1 Improve the inherent overhead 53
6.2.3.2 Provide mechanisms to optimize the instrumentation code 53

Bibliography **55**

A Performance results **59**

List of Tables

3.1 Call types and their equivalent message sends 18
3.2 Object operations and their equivalent message sends 18
3.3 Cache states 21
3.4 Cache events 22
3.5 Cache transition conditions 23
3.6 Object representation operations and their interface 26

A.1 Inherent overhead of Photon 59
A.2 Execution speed slowdown of Photon with deactivated send caches 60
A.3 Performance of the V8 benchmark suite executed by Photon without instrumentation . . 60
A.4 Execution speed slowdown of Photon with a simple instrumentation 61
A.5 Execution speed slowdown of Photon with a fast instrumentation 61

List of Figures

1.1 Components of Photon virtual machine and the features they provide, implement and use 3

3.1 Cache states and transitions 22
3.2 Representation for objects 24
3.3 Representation for special objects 25

4.1 Modification channels 31
4.2 Call graph for example code 34

5.1 Inherent overhead of Photon 43
5.2 Ratios of operations for V8 benchmarks 44
5.3 Slowdown and ratio of **`this`** operations 45
5.4 Execution speed slowdown of Photon with deactivated send caches 46
5.5 Performance of the V8 benchmark suite executed by Photon without instrumentation . . 47
5.6 Execution speed slowdown of Photon with an instrumentation 49

Chapter 1

Introduction

Run-time instrumentation of JavaScript (JS) is currently used for widely different purposes. Notable examples are automatic benchmark extraction from web applications [27], empirical data gathering about the dynamic behavior of Web applications [28] and access permission contracts enforcement [11]. All these examples require instrumentation of object operations, such as property accesses, updates and deletion. They also require instrumentation of all function calls made to global functions, object methods, or function references, either directly or indirectly through their `call` or `apply` method.

The standard semantics of JavaScript has no reflexion feature that completely covers those use cases. Object-method and global-function calls can be wrapped in closures to provide pre- and post-call instrumentation, providing a partial solution. However, direct calls to function references and object operations cannot be intercepted. To work around this limitation, two main approaches are usually employed.

In the first approach, a production virtual machine (VM), usually written in C++, is modified to provide hooks on selected operations. It is increasingly complex on modern JS VMs because those VMs are optimized for performance. The resulting instrumented VM then becomes a burden to maintain up-to-date with its evolving counterpart.

In the second approach, an *ad hoc* source-to-source translator and run-time library are written for the problem at hand and the system is expected to run on top of a production VM. Depending on implementation choices, it can be more or less complicated to guarantee that instrumented objects cooperate well with the rest of the system.

In both cases, instrumentation is achieved at a significant performance cost. On one hand, modifications of production VMs usually are done on an interpreter because the instrumentation hooks can break invariants assumed throughout the JIT-compiler implementation. The modified VM will therefore at best run at interpreter-level speed. On the other hand, source-to-source translations can break the performance advantage of having a JIT-compiler by producing code that is hard for the JIT-compiler to optimize.

In this dissertation, we present Photon, a framework based on *virtual machine layering*. It is inspired by previous work on metaobject protocols [15] but has been adapted to work efficiently on modern JS VMs and provides a better separation of concerns than *ad hoc* source-to-source translations, while performing the source-to-source translation at run time. The optimization techniques presented in this dissertation make Photon run 19% faster on average than a state-of-the-art interpreter. We argue that it makes our approach efficient enough to replace instrumentations that were previously targeted at interpreters on production VMs.

This dissertation therefore shows that *virtual machine layering* is a competitive alternative approach to VM instrumentation for run-time monitoring of object operations and function calls. It makes the following contributions:

- An object representation that efficiently specializes its behavior at run time by relying on the method caching behavior of the underlying host VM (Section 3.2);
- A message-sending optimization that specializes a reified operation to its call site by using the underlying host VM fast global function calls (Section 3.1.3);
- A prototype JavaScript implementation, Photon, comprising both a runtime library (around 1700 lines of code) and JavaScript-to-JavaScript compiler.

1.1 Overview

In a conventional JS setting, an application runs over a high-performance host VM. In the case of a *metacircular* VM, an additional VM layer is inserted between the application and the host VM. This layer can be a *full* or a *differential* implementation. In a full implementation, the metacircular VM provides all functionalities of the source language. In a differential setting, however, the metacircular VM only implements parts of the required functionality, and delegates the remaining operations to the underlying host VM. Our approach follows a differential strategy. Object operations are handled by one of the layers introduced by Photon while primitive operations are handled by the host VM.

This section discusses the objectives that guided the design decisions, followed by an overview of the Photon VM and its components.

1.1.1 Design goals

Our design aims to achieve the following properties:

- **Isolation**: The application is isolated to avoid any interference with instrumentation code, while still allowing an instrumentation to fully inspect and modify the application state.

- **Flexibility**: The semantics of object operations and function calls are exposed as methods so that they can be extended or completely redefined, as required by the instrumentation.
- **Dynamism**: The semantics of object operations and function calls can be redefined while the application is running to allow instrumentations to precisely control the start and duration of their monitoring (e.g., only after the startup phase of the application).
- **Abstraction**: Low-level details, mostly related to performance optimizations, are encapsulated by specific layers to simplify the definition of instrumentations.
- **Performance**: Since native features provided by the host VM are often implemented very efficiently, our implementation reuses them when possible (e.g., the scope chain and control-flow operations). Furthermore, Photon leverages the performance of some host feature (e.g., fast global function calls) to perform optimizations that reduce the overhead introduced by the abstraction layers.

1.1.2 Overview of the Components

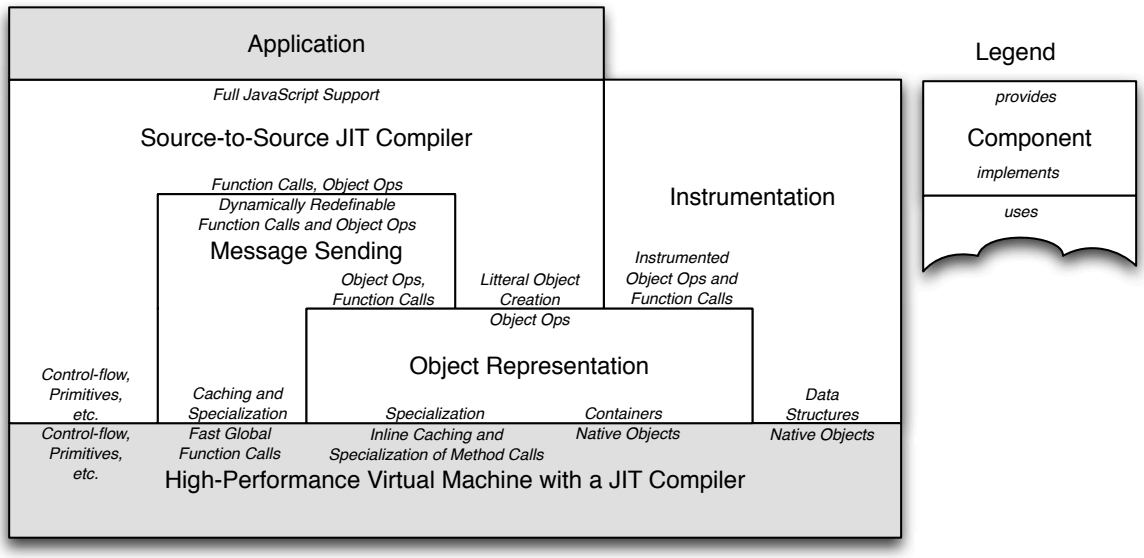

Figure 1.1: Components of Photon virtual machine and the features they provide, implement and use

In a conventional JS setting, we can view an application as running over a host high-performance VM. The metacircular VM approach adds another layer between the two, allowing instrumentations to target

the middle layer instead of the host VM. Figure 1.1 explicits the vertical interaction between layers by naming components and detailing which features are implemented using features of components below it as well as the feature they provide to the layer above. Note that the diagram represents a structural view of component interactions and not the execution model of the application. For example, after JIT compilation, the application code directly runs over the host VM, albeit in a different form, although it is shown as being over the source-to-source JIT compiler. Note that no meaning is associated to horizontal proximity.

We present each component, in a bottom-up fashion, in terms of the abstractions they provide and the key features of their implementation.

1.1.3 Bottom-Up Overview

Object representation. The object representation is the implementation of JS objects (including functions) from the point of view of the Photon VM. For efficiency, it uses native objects as property containers. The native objects are proxied with a second native object to provide encapsulation of invariants of the implementation in proxy methods. It has the benefit of simplifying the implementation of instrumentations because it abstracts implementation details required for performance. It also provides object-specific instrumentation information to be stored on a proxy without risk of interference with the application properties. Object representation operations can be specialized to certain classes of objects for performance, such as indexed property accesses on arrays.

Message Sending. The message-sending layer builds on top of the object representation to provide dynamically redefinable object operations and function calls. This extra level of indirection is used to specialize those operations at their call site, depending on the call site data available, such as argument types and values, as well as the instrumentation state of the VM. It provides dynamic optimization of uninstrumented and instrumented operations. By implementing specialized operations as global functions, the underlying VM will further specialize their behavior and even inline the operations in-place when possible. The invariants implied by the implementation of the message-sending layer are encapsulated in the object representation operations.

Source-to-Source Compiler. The source-to-source compiler translates the original JavaScript code to use the run-time environment of Photon. Non-reified elements, such as control-flow operations and primitive values and operations are preserved. Object operations and function calls are translated to message sends and become dynamically redefinable. Litteral object creation is translated to use the object representation. The source-to-source compiler is in JavaScript and is therefore available at run-time. By staging it in front of every call to `eval`, it effectively provides a JIT compiler to Photon.

Instrumentation. An instrumentation can redefine the behavior of object operations and function calls by replacing the corresponding method on a root object with an instrumented version using the object

representation operations. The ability to completely replace a method provides maximum flexibility to instrumentation writers compared to being limited to a specific event before and after an operation. An instrumentation is written at the same level as the VM, which means that it has access directly to the execution environment of the VM and can use native objects as data structures.

1.2 Background material

To complement the introduction and overview, we elaborate on the specificities of the JavaScript language and our definitions of object model and function calls.

1.2.1 JavaScript

JS is a dynamic language, imperative but with a strong functional component, and a prototype-based object system similar to that of Self.

A JS object contains a set of properties (a.k.a. fields in other OO languages), and a link to a parent object, known as the object's prototype. Properties are accessed with the notation `obj.prop`, or equivalently `obj["prop"]`. This allows objects to be treated as dictionaries whose keys are strings, or as one dimensional arrays (a numeric index is automatically converted to a string). When fetching a property that is not contained in the object, the property is searched in the object's prototype recursively. When storing a property that is not found in the object, the property is added to the object, even if it exists in the object's prototype chain. Properties can also be removed from an object using the **`delete`** operator. JS treats global variables, including the top-level functions, as properties of the global object, which is a normal JS object.

Anonymous functions and nested functions are supported by JS. Function objects are closures which capture the variables of the enclosing functions. Common higher-order functions are predefined. For example, the `map`, `forEach` and `filter` functions allow processing the elements of an array of data using closures, similarly to other functional languages. All functions accept any number of actual parameters. The actual parameters are passed in an array, which is accessible as the `arguments` local variable. The formal parameters in the function declaration are nothing more than aliases to the corresponding elements at the beginning of the array. Formal parameters with no corresponding element in the array are bound to a specific undefined value.

JS also has reflective capabilities (enumerating the properties of an object, testing the existence of a property, etc.) and dynamic code execution (`eval` of a string of JS code).

1.2.2 Object model

An object model is a set of object kinds, their supported operations and the time at which those operations can be performed. Examples of object kinds are arrays, numbers, associative arrays and classes. Examples of operations are object creation, addition, removal or update of properties as well as modification of the inheritance chain. Examples of time for performing operations are *run time*, when a program is executing, *compile time*, when a program is being compiled or *edit time*, when a program's source code is being modified. An object model structures programs to obtain properties on the resulting system such as security, extensibility or performance. Most object models for programming languages provide an inheritance mechanism to facilitate *extensibility* by allowing an object to be incrementally defined in terms of an existing object.

Different languages have different object models. *Class-based* object-oriented languages use *class* objects to describe the properties and the inheritance chain of *instance* objects. For example, C++ class objects exist only at compile time. Property values can be updated at run time. Properties can only be added or removed at edit time and all their accesses are verified at compile time. Java uses run-time objects to represent classes and allows new classes to be added at run time. Classes cannot be modified at run time unless their new definition is reloaded through the debugger API. Ruby class objects exist at run time and new properties can be added also at run time. *Prototype-based* object-oriented languages forgo the difference between class and instance objects. Objects and their inheritance chain are defined directly on objects. Self and JavaScript object properties can be added or removed at run time.

Message sending is an operation that invokes a behavior on an object by executing a program associated to a given message. A message can exist at run time and be an object like any other or it can exist only at compile time. Message sending *decouples* the intention of a program from its implementation by adding a level of indirection between the invocation and the execution of a given behavior. This indirection allows the behavior to change during a program's execution. Therefore, it is a source of *dynamism*.

Some languages call the program associated with a message, a *method implementation*, the message, a *method name* and the act of sending a message, *method calling*. This terminology is closely tied to an implementation strategy where the implementation is a function, the method name is a compile-time symbol and the method call is a lookup followed by a synchronous function call. We use the message-sending terminology because it is more abstract.

A message-sending object model is an object model that takes message sending as its primitive operation. It defines every other operation in terms of message sends. By doing so, all other operations of the object model become dynamic, i.e. they can change at run time. By being based on a single message sending primitive, the optimization effort can be focused on this primitive and run-time information can be used to specialize the behavior invoked, providing an opportunity for *performance*.

1.2.3 Function-calling protocol

The function-calling protocol is a contract between a caller and a callee function and defines what operation each should perform before and after a call. For implementers, it usually refers to the way arguments are passed between the caller and the callee and who is responsible for clean-up of shared data structures such as the stack.

In our system, there is no difference between the passing of arguments of the host VM and the layered VM. However, to react to function-calling events, there is a need to have a place to define operations to be executed before and after calls. We will therefore refer to the function-calling protocol as the operations that are performed before and after a call.

The `call` method on functions in JS is already a form of reified calling protocol. Our design exploits it to change the behavior of *all* function calls when it is modified, which is not the case in the current standard.

1.3 Outline

The remainder of this dissertation covers the following subjects:

- Chapter 2 presents previous work on flexible computer systems.
- Chapter 3 explains the design of the VM with a particular emphasis on the message-sending foundation and the object representation.
- Chapter 4 presents use cases in modifying the VM that are either difficult or impossible to do with existing JS implementations.
- Chapter 5 compares our implementation with a state-of-the-art implementation to establish the overhead of providing the flexibility and show suitability to replace existing instrumented interpreters.
- Finally, the conclusion in Chapter 6 explains the current limitations of the system and sketches ideas for further work using our current results

Chapter 2

Previous Work

"Those who cannot remember the past are condemned to repeat it."
— George Santayana, 1863 – 1952

The work described in this dissertation started as an exploration of the possibilities offered by an object model based on the open, extensible object model by Piumarta and Warth [23]. In this chapter, we first trace back the original sources of core ideas that eventually influenced our implementation by organizing them around four themes, *openness*, *extensibility*, *dynamism* and *performance* and by focusing on issues pertaining to the object model and function-calling protocol. We finally review other existing systems that perform dynamic instrumentation of JavaScript and compare them to Photon.

2.1 Historical roots

The previous work is presented in chronological order to provide a sense of evolution and temporal distance. Throughout the exposition, we make explicit the elements that inspired our work and we contrast the elements that are different.

2.1.1 Openness

The meaning of open depends on what is being qualified. Perhaps the most common usage is open *source*. It refers to accessibility of the text representation, or source code, that was used to produce the system. Accessibility of the source code and development tools allows a motivated person to modify the text representation and generate a new version of the system incorporating the change. However, it says nothing about the nature and the number of changes that are be required to obtain a desired behavior.

We use the term open as an implementation qualifier. It therefore means that the behavior of the system can be modified by first-class data structures, e.g. closures. The range of behavior modifications

allowed by an implementation characterizes the degree of openness.

The earliest account of the need for open implementations seems to be made by Shaw and Wulf in 1980 [29]. In it, they argue that the approach taken to define abstract data types should also be used to define the behavior of language operations, namely to separate the *specification* from its *implementation*. They identify storage layout, variable handling, operator semantics, dynamic storage allocation, data structure traversal and scheduling and synchronization as areas ripe to benefit from being opened. Compared to our focus on opening the object model operations and function-calling protocol, these elements are closer to the machine execution model than to the language's semantics.

Smith worked concurrently on similar areas with a different approach. He first asked how a system could reason about its own inference processes. This was an inherently philosophical question. In his PhD dissertation in 1982 [30], he answered it, not only by clarifying the notions behind reflectivity, but also by showing how an interpreter for Lisp could be built to satisfy those notions. His work planted the seeds for an influential line of research. Compared to Shaw and Wulf work, it directly addressed the semantic implications of opening implementations while they merely suggested that some elements should not be bound early without giving any indication on how to define the semantics of a programming language to do so. The notion of reflection is also stronger because it implies the ability to modify the implementation from within the language itself. Among the notions Smith introduced, we use the *causal connection* criteria to think about JavaScript semantics in section 3.1.1.2. This criteria says that a data structure is causally connected to the behavior of a system if changing the data structure changes the behavior of the system.

In a landmark OOPSLA paper in 1987 [20], Maes showed how to bring reflection to object-oriented programming languages through meta-objects. These objects were shown to encapsulate various elements of the language implementation. Later in 1991, Ramano Rao introduced the open implementation term to replace the reflection architecture term that was previously in use to emphasize that not only language implementations can be open but also applications [24]. Then, a methodology for designing open implementations was suggested by Maeda and al. [19], first by taking an existing closed implementation and then by progressively opening all the elements of interest. This line of work took the conceptual underpinnings of reflection and integrated it with other language developments at the time, showing that it could be used in practice to build not only programming languages but whole systems around those principles. In the same spirit, our work applies the open implementation idea to a mainstream language, JavaScript, and solves the associated design and performance issues associated with it.

2.1.2 Extensibility

The term extensible means that a VM's components can be independently modified or replaced, that new components can be supported without having to modify existing components and they can be

incrementally defined in terms of existing components. Encapsulation and modularity covers a subset of that definition.

A major breakthrough in structuring the definition of computer systems was the introduction of the notion of *inheritance* in 1966 by Dahl, Myrhaug and Nygaard in the Simula 67 language [8]. The language had the notion of classes and objects and new classes could be defined in terms of existing classes through its inheritance mechanism. The language had a major influence on the design of Smalltalk [13] and C++ [31].

Another major idea for extensibility was the notion of *encapsulation*, initially called *information hiding* by Parnas in 1972 [22]. In his paper, he argued that instead of decomposing systems by subroutines performed, they should be decomposed such that modules encapsulate key design decisions, in a way that is as independent as possible from other decisions. Later Dijkstra argued for a similar idea which he named *separation of concerns* and applied it not only to system design but also problem-solving and general thinking. The first occurrence of the term seems to be found in 'A Discipline of Programming' [10]. The object-oriented community finally adopted encapsulation to describe the idea, when applied to computer systems.

In the object-oriented community, a class-based approach to object decomposition was favored until Lieberman proposed in 1986 [18] to use a prototype-based approach to implement shared-behavior. This latter mechanism was shown to be more expressive since it could be used to implement an equivalent class-based mechanism but the opposite was not possible. This had a major influence on the design of Self [35] which later influenced the design of JavaScript.

The object representation presented in section 3.2 exploits prototype-based inheritance and a method-based behavior definition that allows encapsulation of design decisions made for optimizations. It allows instrumentations to be written for our system with minimal knowledge about its inner working. Our work does not contribute new ideas for extensibility but uses powerful existing ones to simplify its implementation and evolution.

2.1.3 Dynamism

Dynamism has seen an increasing usage through the popularization of dynamic languages. Again, the actual meaning of dynamism in dynamic languages is fuzzy. The common thread however is the presence of *late-binding* of some aspects of the programming language, namely deferring until run time the actual computation required to implement a given functionality.

There is an overlap between the work presented in the previous section and this one because ideas for extensibility evolved in parallel with ideas for dynamism.

The earliest account of dynamic strong typing seems to be in the initial Lisp paper by McCarthy in 1960 [21]. In it, the type of variables was enforced (strong typing) at run time (dynamic typing).

Later, polymorphism through message-sending, or the ability of a given identifier to invoke different behavior during execution, was pioneered both in its asynchronous version in the Actor model by Hewitt in 1973 [12] and in its synchronous version by the Smalltalk crew, led by Kay between 1972 and 1976 [13]. The main difference between the asynchronous version and the synchronous version is that in the former, the flow of control returns to the sender before the message has been processed while it waits for the message to be processed in the latter. In both cases, message-sending is the primitive operation on which all the other operations are based.

Going back to Smith in 1982 [30], in his work on reflection, we can also note that the emphasis was put on the ability to reflect upon the dynamic behavior of programs, which would later be called behavioral reflection although the term Smith used was procedural reflection.

Finally, one important characteristic of prototype-based inheritance, as explained by Lieberman in 1986 [18] is its dynamic nature. The prototype of an object is known and can change during program execution.

JavaScript uses dynamic strong typing and prototype-based inheritance. We exploit these characteristics for dynamic instrumentation. In our work, the idea of using message-sending as a foundation was inspired by Smalltalk and the dynamic nature of the instrumentation we perform owes a great deal to the behavioral reflection work. We use reflection capabilities in chapter 4 to show how to change the semantics of object model operations.

2.1.4 Performance

Dynamic languages have had the reputation of being slow and inefficient. This is changing as mainstream languages incorporate more dynamic features and considerable engineering efforts are targeted at improving their performance. Apart from optimizations deriving from idioms of a particular language, two historical papers provide the ground works we used.

In 1984, Deutsch an Schiffman introduce the inline cache technique to optimize method dispatch on class-based object-oriented languages [9]. The technique uses dynamic code patching to memoize the class of the receiver to store the lookup result at the call site for faster later execution. In 1989, Chambers and al. generalized the technique to prototype-based languages by introducing the concept of *map*, an implicit class constructed by the implementation as objects are created and modified during execution [7].

Both of these techniques were inspirations for the optimizations we explain in section 3.1.3.

2.2 Instrumenting JavaScript VMs

Kikuchi et al. reported on the real-world applicability of instrumenting JS to enforce security policies [16]. They discuss instrumentation of web applications by intercepting and rewriting incoming JS code in

web pages by a proxy server operating in-between a client browser and a web server. Their work is complementary to ours. Their rewriting strategy could be adjusted to use Photon's execution model and applied to other problems than security policy application. Their rewriting strategy also targeted object model operations and function calls and they reported an impact of less than a factor of two in execution time on web applications, such as GMail. However, these results were obtained in 2008, at the beginning of JIT compilation strategy adoption in web browsers. Given our performance evaluation, a higher ratio would be expected on modern VMs.

Sandboxing frameworks for JS, such as Google Caja [1], BrowserShield [26] and ADSafe [2] guarantee that guest JS code cannot modify the host JS environment outside of a permitted policy. We focus here on Google Caja as a representative candidate. The Caja sandbox provides a different global object to the guest code and performs a source-to-source translation to ensure that all operations on host objects are mediated by proxies enforcing a user-defined security policy. Photon also provides a different global object to the running application and performs a source-to-source translation to mediate object operations through proxies. However, it is done to simplify reasoning about instrumentations while providing an acceptable level of performance. Our sandboxing strategy does not need to be as stringent, therefore we deem acceptable the possibility of leaking the native objects by accessing the `__proto__` property. Documentation of the run-time performance penalty of their approach is sparse. However, the authors do report a ˜17× slowdown on EarleyBoyer when running over Rhino 1.6.7[1], a somewhat slower JS VM than the Firefox interpreter (in comparison, Photon has a 9.53× slowdown for EarleyBoyer on the Firefox interpreter).

JSBench [27] performs instrumentation of object operations and function calls for recording execution traces of web applications that can be replayed as stand-alone benchmarks. Instrumented operations then call global functions, in a way similar to the esprof prototype we built (Section **??**). Its authors report an impact on performance that is "barely noticeable on most sites" even when running in interpreter mode on Safari. JSBench instrumentation is specially tailored to the task of recording benchmarks while Photon aims to be a general instrumentation framework.

The idea of using aspect-oriented programming for profiling tasks has been successfully used in the past, although some limitations of the model have been identified (e.g., [6]). AspectScript [34] has similar aims as Photon, namely providing for JS a general interface for dynamic instrumentation of object operations and function calls. It uses a source-to-source translation scheme with a single *reifier* primitive which is analogous to our *message-sending* primitive. They document a scoping strategy that allows targeting a single object or function, instead of all objects globally, which is more flexible than Photon's current strategy. We believe that Photon could be extended similarly. Compared to our instrumentation interface, they use the dynamic weaving of aspect formalism instead of our "operations as methods" approach. Because of the use of the aspects formalism, their approach provides better encapsulation

[1] https://code.google.com/p/google-caja/wiki/Performance

of the instrumentation strategy at the expense of flexibility and performance. We executed the latest version of AspectScript against the V8 benchmarks, and found it to be between 10× and 454× slower than Photon on Safari. Additionally, only four of the benchmarks ran without errors.

Js.js [33] is a JS port of the Firefox interpreter compiled using the Emscriptem C++ to JS compiler. It is intended for sandboxing web applications. The resulting JS interpreter then runs in the browser on top of an existing VM. However, this is a heavy-weight approach with a significant performance overhead. The authors report slowdowns between 100× and 200× for most cases with worst cases of 600× on SunSpider benchmarks. The large slowdown can be attributed to the full simulation of a native interpreter in JS. Photon avoids reimplementing features of the language outside object operations and function calls. The resulting implementation is both faster and simpler to instrument.

Other approaches target the host VM for efficiency reasons. JSProbes [5] is a series of patches to the Firefox interpreter that allow instrumentations to be written in JS and target pre-defined probe points, such as object creation, function calls and implementation events such as garbage collector start and stop events. JSProbes provides much of the same properties as Photon at a much lower execution overhead and with additional information about implementation events that are inaccessible to Photon. It is however less flexible since for the same language operations, JSProbes is limited to a fixed set of predefined events while Photon allows the semantics of operations to be changed. Unfortunately, targeting a VM implementation requires maintenance to follow the upstream modifications. At the time of writing, maintenance of JSProbes has stopped, making the approach unavailable in practice. In a different setting, Lerner et al. explored the requirement for implementing aspect support in an experimental JIT-compiler [17]. They reported a simpler and more efficient implementation than other aspect-oriented approaches. Their work was intended to inform possible ways to open native implementations to instrumentation with an aspect formalism. So far, no production VM implements aspects, which makes this approach unavailable in practice. Photon does not require modifications to the host VM. It therefore does not add to the maintenance cost of production VMs to be usable in practice.

Recently a new low-level implementation of the JavaScriptCore interpreter optimized for performance became available. In our tests on the V8 benchmark suite, the new interpreter is roughly 3× faster than the Firefox interpreter. The performance gains are attributable to using an assembly language. As this new interpreter matures its complexity will likely increase, negating most of the simplicity usually attributed to interpreters. This makes the task of instrumenting the new JavaScriptCore interpreter much harder to achieve and maintain, and consequently, the Firefox interpreter is currently the only JS interpreter that is reasonably fast and sufficiently easy to instrument.

The Narcissus JS in JS interpreter implementation by Mozilla reifies all the language operations of the language. It could be instrumented by inserting hooks in the corresponding case statements of the main interpretation loop. However, compared to Photon, Narcissus is much slower. Unfortunately, none of the V8 benchmarks could be executed. However, on a micro-benchmark stressing the function calling

protocol, it was two orders of magnitude slower than Photon. The overhead can be attributed to the reification of activation records for function calls. In contrast, Photon uses the execution environment of the host VM to provide better performance.

Chapter 3

Design

"There are only two kinds of languages: the ones people complain about and the ones nobody uses."
— Bjarne Stroustrup [1]

In this chapter, we elaborate on the design of the message-sending and object-representation layers because they are the key components of Photon's runtime environment. As introduced previously, the former is the primitive used to reify low-level operations and the latter serves to efficiently isolate the application from the instrumentation code. We first introduce the semantics of the message-sending operation and discuss its efficient implementation. We then introduce our object representation, which takes advantage of the method invocation optimizations performed by the underlying VM. Finally, a brief compilation example is provided to illustrate how JavaScript can be compiled to this runtime environment.

3.1 Message-sending foundation

In our design, the message-sending operation serves to unify all reified operations, namely operations on objects and function calls, around a single mechanism. It facilitates the achievement of correctness, because a single primitive operation needs to be added to the implementation, and performance, because its implementation targets the highest performing operations of the underlying system. The particular choice of message-sending as a foundation was motivated by its tried and true application to Smalltalk and Self with success for the obtention of a dynamic, open and fast implementation.

3.1.1 Semantics

We now see the chosen semantics for the message-sending operation, in steps. We first discuss the basic operation, that closely corresponds to a method call in JavaScript. We then reify the function call to

[1] *Of course, all "there are only two" quotes have to be taken with a grain of salt.* [32]

allow its redefinition. We then reify the memoization operation performed by inline caches to enable functions to specify their cached behavior.

The implementation of message sends uses our object representation but could accomodate other representation choices. Our object representation will be fully explained in the next section. For now, it is sufficient to think of our representation as proxies to native objects with methods for reified operations that provide isolation and encapsulate low-level details necessary for optimization. This section uses the following methods:

- `call(rcv, ..args)`: calls the object as a method with the `rcv` receiver and `args` arguments and returns its return value
- `get(name)`: returns the value of the `name` property on this object and implicitly performs a lookup on the prototype chain
- `unbox()`: returns the native object wrapped by the representation

Unless specified, the pseudo-code follows the regular JavaScript semantics. For presentation simplicity, we use the rest and spread operators (`..args`), to appear in the next version of JavaScript [4], and we omit primitive values, missing values and error handling.

3.1.1.1 Message sending as a method call

The basic version essentially performs a regular lookup followed by a call to the function retrieved with the receiver object. This corresponds to the semantics of a method call in JavaScript:

```
function send(rcv, msg, ..args) {
    var m = rcv.get(msg);
    return m.call(rcv, ..args);
}
```

3.1.1.2 Reifying function calls

In JavaScript, the `call` method on every function reifies the calling protocol and allows a program to call into a function at run time, as if it was done through the direct mechanisms. The exact behavior of a function call should be the same, whether it is called directly or indirectly. However, there is no causal connection between the state of the `call` method and the behavior of function calls. In other words, redefining the `call` method on `Function.prototype` does not affect the behavior of call sites.

We can establish this causal relationship with a slight modification of the send operation:

```
function send(rcv, msg, ..args) {
    var m = rcv.get(msg);
    var callFn = m.get("call");
    return callFn.call(m, rcv, ..args);
}
```

This semantics allows a particular function to be instrumented, simply by redefining its call method.

3.1.1.3 Providing a memoization hook

When designing an instrumentation framework, a choice in the granularity of instrumentation needs to be made. For example, a corse-grained design would use a single method implementation for all operation call sites. For Photon, we chose a fine-grained design, where the implementation of every call site can be different. An operation, instrumented or not, is specialized with the information available at the call site. We expose this functionality to instrumentation writers through a special `__memoize__` method, specific to each reified operation.

Our mechanism allows a method to inspect its arguments and receiver to specialize itself for subsequent calls. The first call is always performed by calling the original function while all subsequent calls are made to the memoized function. A `__memoize__` method on a reified operation is called the first time the operation is invoked to provide the specialized method that will be used on all subsequent calls.

There is an unfortunate interaction between memoization and the reification of the call protocol. Care must be taken because memoization can only occur if the call method of the function has not been redefined. Otherwise, the identity of the function passed to the call method would not be the same. To preserve identity while allowing memoization, the behavior of the cache can be different depending on the state of the `Function.prototype`'s `call` method. If its value is the default one, the identity of the function is not important and memoization can be performed. Otherwise, memoization is ignored.

The extended semantics, including memoization at the call site is the following:

```
var defaultCallFun = root.function.get("call");

function send(rcv, msg, ..args) {
    var m = rcv.get(msg);
    var callFn = m.get("call");

    if (callFn === defaultCallFn) {
        var memFn = m.get("__memoize__").call(m, rcv, ..args);

        if (memFn !== null) {
            ...Store memFn in cache
        }

        return m.call(rcv, ..args);
    } else {
        return callFn.call(m, rcv, ..args);
    }

}
```

3.1.2 Translating reified operations to message-sending

Reifying object operations and function calls is done by translation to corresponding message sends. Table 3.1 summarizes all the different source code occurrences of calls and their equivalent message sends and Table 3.2 explains the translations for object operations.

Table 3.1: Call types and their equivalent message sends

Call Type	Explanation	Equivalent Message Send
Global	Calling a function whose value is in a global variable. Ex: `foo()`	Sending a message to the global object. Ex: `send(global,"foo")`
Local	Calling a function in a local variable. Ex: `fn()`	Sending the `call` message to the function. Ex: `send(fn,"call")`
Method	Calling an object method. Ex: `obj.foo()`	Sending a message to the object. Ex: `send(obj,"foo")`
`apply` or `call`	Calling the `call` or `apply` function method. Ex: `fn.call()`	Sending the `call` or `apply` message. Ex: `send(fn,"call")`

Table 3.2: Object operations and their equivalent message sends

Object Model Operation	Explanation	Equivalent Message Send
Property access	Retrieving the value of a property that might exist or not. Ex: `obj.foo`	Sending the *__get__* message. Ex: `send(obj,"__get__","foo")`
Property assignation	Creating or updating a property. Ex: `obj.foo=42`	Sending the *__set__* message. Ex: `send(obj,"__set__","foo",42)`
Property deletion	Deleting a property that might exist or not. Ex: **`delete`** `obj.foo`	Sending the *__delete__* message. Ex: `send(obj,"__delete__","foo")`
Object litteral creation	Creating an object in-place. Ex: `{foo:42}`	Sending the *__new__* message. Ex: `send({foo:42}, "__new__")`
Constructor creation	Creating an object with **`new`**. Ex: **`new`** `Fun()`	Sending the `__ctor__` message. Ex: `send(Fun, "__ctor__")`

3.1.3 Efficient implementation

The reification of low-level operations of JS with message sends introduces a performance penalty compared to a native execution. Our design takes advantage of optimizations performed by the host VM to efficiently implement reified operations. The core insight behind our implementation comes from seeing global function calls of the host VM both as an optimized calling mechanism and as a dynamically specializable operation. They provide the same ability as code patching in assembly. On the current

version of V8, when the number of arguments expected matches the number of arguments supplied, it opens the possibility of inlining the function at the call site. If the global function changes at a later time, the call site is deoptimized transparently. It is a really powerful mechanism because much of the complexity of run-time specialization is performed by the underlying host. We can simply piggyback on those optimizations.

To motivate the actual optimizing implementation, let's use the following unoptimizing translation of a call to the method `bar` on the object `obj` inside a function `foo`:

```
function foo(obj) {
    send(obj, "bar"); // Equivalent to obj.bar();
}
```

Notice that by specializing the operation to the information available at its call site, a more efficient version can be provided. In this example, the name of the method is constant. If we further suppose that the `call` method on `Function.prototype` has not been redefined, the implementation of a message-send can be inlined and stripped of the generality required to reify function calls. Using the memoization mechanism the resulting call site will be specialized to the following implementation:

```
var codeCache0 = function (rcv, x0) {
    return rcv.get("bar").call(rcv);
};

function foo(obj) {
    codeCache0(obj, "bar");
}
```

In Photon, this specialization is done at run time. Initially, the global variable corresponding to the call site is initialized with a generic function that will be replaced by the specialized version during the first call to the reified operation. The next listing shows the actual implementation in the initial state of the cache:

```
var initState = function (rcv, dataCache, ..args) {
    var dataCacheName = "dataCache" + dataCache[0];
    var codeCacheName = "codeCache" + dataCache[0];
    var msg = dataCache[1];

    var callFn = method.get("call");
    if (callFn === defaultCall) {
        var memMethod = method.get("__memoize__").call(method, rcv, args, dataCache);

        if (memMethod !== null) {
            global[codeCacheName] = memMethod.unbox();
            // Track memoization sites for invalidation
            tracker.addCacheLink(msg, dataCache[0], dataCache);
            return method.call.apply(method, [rcv].concat(args));
        }

        // Use a specialized version of a message send for a given message name,
        // also avoiding the call indirection introduced the function calls
        // reification
        global[codeCacheName] = memNamedMethod(msg, args.length);
        return method.call.apply(method, [rcv].concat(args));
    } else {
        return callFn.call.apply(callFn, [method, rcv].concat(args));
    }
};

var codeCache0 = initState;
var dataCache0 = [0, "bar"];

function foo(obj) {
    codeCache0(obj, dataCache0);
}
```

Executing the code will eventually replace `codeCache0` with the expected method, corresponding to the second return statement in the `initState` function.

3.1.3.1 Cache states and transitions

The previous section has shown two different states of a message send call site, used to optimize subsequent calls to reified operations. In this section, we fully define the behaviors of the inline caches and the conditions that trigger those behaviors. We use a state machine formalism to present the different behaviors associated with inline caches and the triggers that are responsible for the transitions between those behaviors. In our formalism, due to the nature of synchronous message sends, a state transition occurs before the event has been fully processed. However, the processing of the event is not influenced by it.

To simplify the tracking of invariants, we decided to always perform lookups for regular method calls. By making the lookup operation as close as possible to the native operation of JavaScript, it allows the underlying VM to optimize it. In our design, the idea is to delegate the regular method call to the object representation. The other important operation was to allow specialization of object model operations,

which is critical for speed. This is done by memoizing a specialized version in the inline cache. We therefore ended up with two states in addition to the initial state of the cache, as explained in table 3.3.

Table 3.3: Cache states

Cache states	Explanation
Initial State	Perform a regular send.
Regular method call	Lookup method then call.
Memoized method call	Method specific behavior. The memoized method is responsible for maintaining invariants.

Transitions between states happen on message sends and object model operation events. An insight was to realize that we could underapproximate the tracking of invariants and conservatively invalidate more caches than what would minimally be required. As long as the operations triggering the invalidation of caches are infrequent, the performance impact should be minimal. We therefore track method values cached in memoized states by name without consideration for the receiver object. If a method with the same name is updated on any object, all caches with a given message name are invalidated. Also, if the `call` method on the `Function.prototype` object or any method with the `__memoize__` name is updated, *all* caches are invalidated. This way, we can only track caches associated with names. The upper bound on memory usage for tracking information is proportional to the number of cache sites.

There is no state associated with a redefined `call` method. In that particular case, all caches stay in the initial state and a regular message send is performed. Figure 3.1 summarizes those elements in a state diagram. A more detailed explanation of every event and transition conditions are given in table 3.4 and table 3.5.

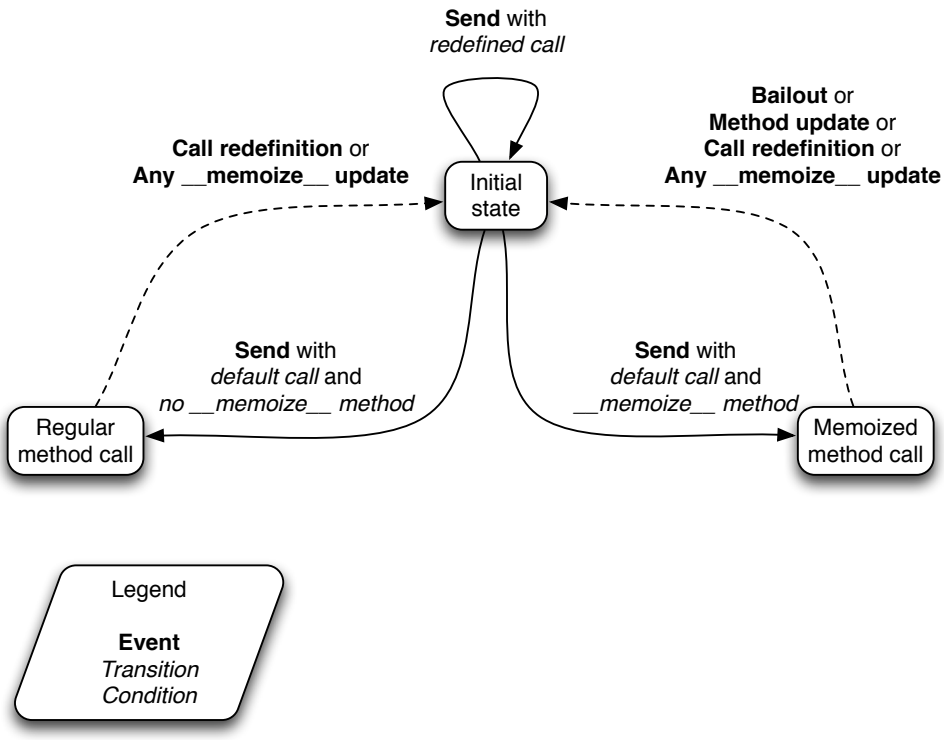

Figure 3.1: Cache states and transitions

Table 3.4: Cache events

Cache Events	Explanation
Send	A message is sent to a receiver object.
Call redefinition	The `call` method on `Function.prototype` is redefined.
Any memoized redefinition	Any `__memoize__` method is being redefined.
Bailout	A run-time invariant has been violated.
Method update	An object's method is being updated.

Table 3.5: Cache transition conditions

Cache Transition Condition	Explanation
Default call	`Function.prototype call` method is the same as the one initially supplied.
Redefined call	`Function.prototype call` method is different than the one initially supplied.
No `__memoize__` method	No method named `__memoize__` has been found on the method to be called.
`__memoize__` method	A method named `__memoize__` has been found on the method to be called.

3.2 Object representation

The object representation in a virtual machine is a core aspect that has a major influence on its flexibility and performance. The following object representation has been designed to encapsulate the invariants of our implementation while enabling piggybacking on the underlying VM inline caches for performance. We believe that our particular usage of the underlying object model dynamism is new and could serve as a basis for other language implementations seeking performance on modern JS VMs.

Two insights led to the current design. First, on a well optimized VM, the most efficient implementation of a given operation in a metacircular implementation is frequently the exact same operation performed by the host VM. Second, method calls on JavaScript VMs are usually really fast. Therefore, Photon provides operations as method calls on objects whose internal representation is as close as possible to the host object being implemented.

The core idea is to associate a proxy object to every object in the system and have the proxy prototype point to the parent object's proxy. Photon optimizes the object representation operations at runtime by attaching specialized methods at the appropriate places on the proxy prototype chain.

The object representation is presented for native object types. A refinement is presented for special objects to preserve performance. We then see how the object representation is used to specialize methods for the number of arguments found at their call site.

3.2.1 Representation for objects

The first and simplest object type in JavaScript is the object. It has properties and a prototype. A proxy object has a reference to the native object to intercept every operation that goes to the object.

The prototype chain of proxies mirrors the object prototype chain. A JavaScript implementation, with `Object.prototype`, as the root of all objects, is illustrated in Figure 3.2.

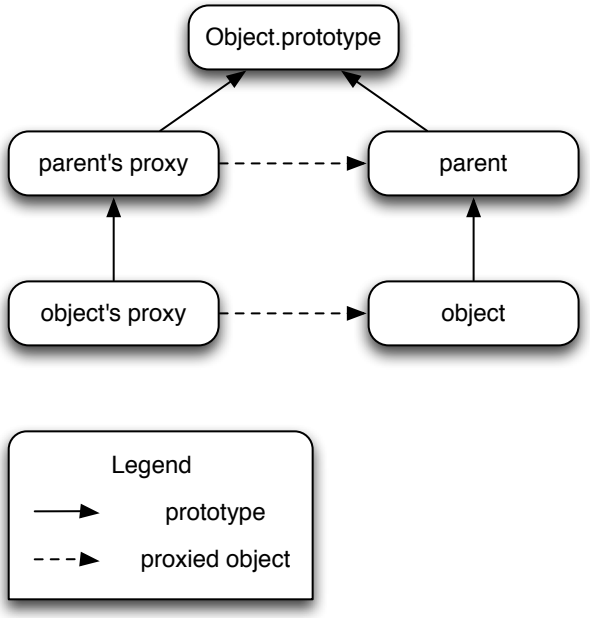

Figure 3.2: Representation for objects

An advantage of this representation is that property accesses can be implemented directly as a native property access to the proxied object. It allows the host VM to do lookup caching. It even works for reified root objects, in this example, by considering `parent` the `Object.prototype` of all objects of the metacircular VM.

However, it does not work well with native types that can be created literally such as arrays, functions and regular expressions. These would require their prototype to be changed to another object at the creation site, ruining structural invariants assumed by the host VM. For those objects, the original native prototype is maintained and in cases where a lookup needs to be performed, it is done explicitly through the proxy prototype chain. This is illustrated with arrays in Figure 3.3.

Given this structural representation for all object types, we can now define all object operations as methods on proxy objects as explained in Table 3.6. Remark that given the current JavaScript *de facto* standard of accessing the prototype object with the `__proto__` name, if proper care is not taken in the property access method, the proxied object will be returned instead of the expected proxy object of the parent. [2]

[2]For this particular reason, we would advocate for implementations to expose the prototype of an object through a

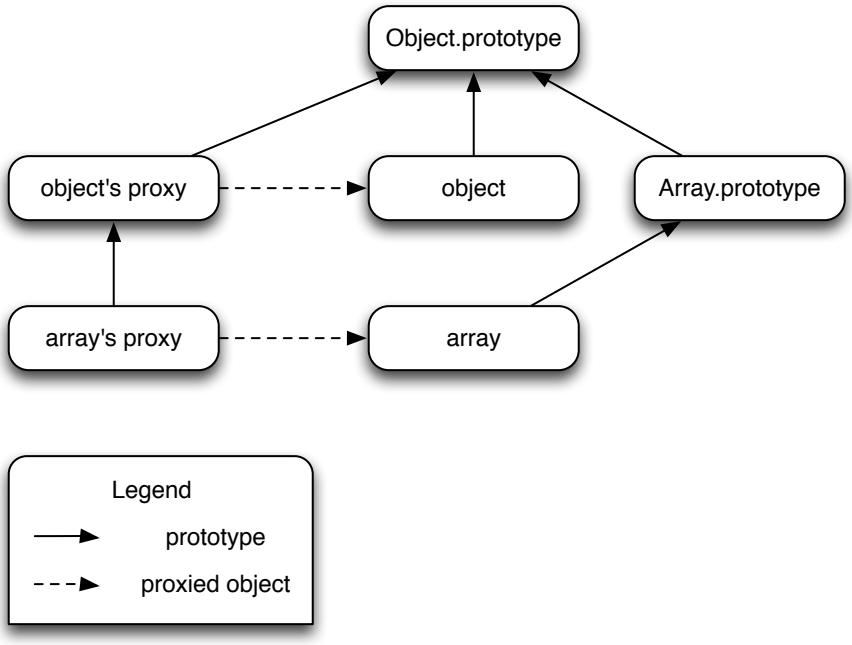

Figure 3.3: Representation for special objects

Proxies can adapt to dynamic circumstances by adding specialized methods at run time, which can be used for performance gains. The next section shows how Photon exploits this possibility to specialize operations to a fixed number of arguments. To avoid name clashing and ease reading we adopt the convention that specialized methods share the same prefix as basic methods with the type or value information and or number of arguments following in the name.

method call instead of a property. We could still preserve backward compatibility for literal object definitions but once the object is created, the prototype should not be accessible or modifiable through the `__proto__` property.

Table 3.6: Object representation operations and their interface

Operation	Explanation	Interface
Property access	Retrieving the value of a property that might exist or not. Ex: `obj.foo`	`get(name)` Ex: `obj.get("foo")`
Property assignation	Creating or updating a property. Ex: `obj.foo=42`	`set(name, value)` Ex: `obj.set("foo",42)`
Property deletion	Deleting a property that might exist or not. Ex: **`delete`** `obj.foo`	**`delete`**`(name)` Ex: `obj.`**`delete`**`("foo")`
Property test	Test if a property name is present or not. Ex: `obj.hasOwnProperty("foo")`	`has(name)` Ex: `obj.has("foo")`
Object creation	Creating an object from a parent object. Ex: `Object.create(parent)`	`create()` Ex: `parent.create()`
Call	Call the object as a function. Ex: `fun()`	`call(rcv, ..args)` Ex: `fun.call(global)`
Box	Returns the proxy of an object if applicable.	`box()` Ex: `obj.box()`
Unbox	Returns the proxied object.	`unbox()` Ex: `obj.unbox()`
Prototype access	Returns the prototype of an object. Ex: `obj.__proto__`	`getPrototype()` Ex: `obj.getPrototype()`
Prototype update	Sets the prototype of an object. Ex: `obj.__proto__ = parent`	`setPrototype(parent)` Ex: `obj.setPrototype(parent)`

3.2.2 Fixed arguments number specialization

The object representation design does not require a special calling convention for functions. However, for maximum performance gains in JavaScript, we would like to avoid using the `call` and `apply` native methods. We can do it by globally rewriting every function to explicitly pass the receiver object. This way, a specialized call operation can simply pass the references to the native function. An example implementation for a `call` operation, specialized for one argument in addition to its receiver, is:

```
var fun = FunctionProxy(function ($this, x) { return x; });

fun.call1 = function ($this, arg0) {
    return this.proxiedObject($this, arg0);
};
```

Specializing one proxy operation requires to specialize *all* objects for that particular operation to ensure that whatever receiver, function or object is called at a given site, a proper operation is supplied. Fortunately, the object-oriented nature of our chosen representation makes it easy. Only root proxies need to have an additional method and all other proxies then implement the specialized operation.

3.3 Compilation example

To wrap up and see how those different elements work together in practice, let's finish with a brief compilation example. We use the following example:

```
(function () {
    var o = {};
    o.foo = function () { return this.bar; };
    o.bar = 42;

    for (var i = 0; i < 2; ++i) {
        o.foo();
    }
})();
```

In this example, an anonymous function is called right after being created to provide a lexical scope, which means that the `o` and `i` variables are local to the function. In this scope, we create an `o` empty object, which has the root object of the metacircular VM for prototype. Then, this object is extended with a `foo` method. This method returns the `bar` property of the receiver object. We then create and initialize the `bar` property of the `o` object. Finally, we call the `foo` method two times to give it the chance to specialize the call site, both of the `foo` call and the `bar` property access inside the `foo` method.

We first see how compiling this example to message sends uses the inline caching idiom. The compiled code has been weaved with the original code in comments to allow the reader to follow easily:

```
(codeCache0 = initState);
(dataCache0 = [0, "__new__",[]]);
(codeCache1 = initState);
(dataCache1 = [1,"__get__",["this","string"]]);
(codeCache2 = initState);
(dataCache2 = [2,"__new__",[]]);
(codeCache3 = initState);
(dataCache3 = [3,"__set__",["get","string","icSend"]]);
(codeCache4 = initState);
(dataCache4 = [4,"__set__",["get","string","number"]]);
(codeCache5 = initState);
(dataCache5 = [5,"foo",["get"]]);
(codeCache6 = initState);
(dataCache6 = [6,"__new__",[]]);
(codeCache7 = initState);
(dataCache7 = [7,"call",[]]);

// (function () {
(codeCache7((codeCache6(
    root.function, dataCache6,
    (new FunctionProxy(function ($this,$closure) {
        var o = undefined;
        var i = undefined;
        // var o = {};
        (o = (codeCache0(root.object, dataCache0, (root.object.create())))));
        // o.foo = ...
        (codeCache3(o, dataCache3, "foo",
        //     ... function () { return this.bar; };
            (codeCache2(
                root.function, dataCache2,
                (new FunctionProxy(function ($this,$closure) {
                    return (codeCache1($this, dataCache1, "bar"));
                }))))));
        // o.bar = 42;
        (codeCache4(o, dataCache4, "bar", 42));
        // for ...
        for ((i = 0); (i < 2); (i = (i + 1))) {
            // o.foo();
            (codeCache5(o, dataCache5));
        }
    }))
// })();
)), dataCache7, root.global));
```

In addition to what has been discussed in this chapter, this example exhibits additional implementation details:

- *Type information in data caches*: During compilation, known types which directly correspond to abstract syntax tree nodes are preserved. It allows the runtime to exploit stable information. For example, in `dataCache1`, the `"string"` type allows the runtime to know the property access is to a constant string name.

- *Root objects are different from the host root objects*: `root.function`, `root.object` and `root.global` virtualize the object model root objects to avoid interference with the host objects.

- *Functions have an extra `$closure` parameter*: This extra parameter is used to pass the proxy to the function to the code or dataCache information for the implementation to send the cache state to the cache function behavior.

Apart from those additional details, the compilation conforms to the explanation given in this chapter. The environment is used as such, without reification, objects are proxied or created from object representation methods.

After execution, the inline caches at `codeCache1` and `codeCache5` are respectively in a memoized state and a regular method call state, which correspond to the following behaviors:

```
codeCache1 = function ($this,dataCache,name) {
    return $this.get(name);
}

codeCache5 = function($this,dataCache) {
    return $this.get("foo").call0($this);
}
```

In the last case, we can see that the `call` method has been specialized for no arguments, exploiting optimization opportunities offered by our object representation. This example therefore summarizes the unification of object model operations to message sends, their efficient implementation and a novel object representation that can dynamically adapt itself to information available at runtime.

The next chapter shows how we can use the added flexibility for instrumentation and runtime invariant monitoring.

Chapter 4

Flexibility

"You must be shapeless, formless, like water. When you pour water in a cup, it becomes the cup. When you pour water in a bottle, it becomes the bottle. When you pour water in a teapot, it becomes the teapot. Water can drip and it can crash. Become like water my friend."
— Bruce Lee

In this section, we explore how to use Photon for modifications that would be difficult or impossible to do using the standard semantics of JS and that might be complicated if we tried to modify the host VM. Our first motivation was to facilitate the dynamic instrumentation of JS programs. We will also see some modifications that go beyond instrumentation tasks to illustrate that the added flexibility can be used for more general tasks.

We will see different examples modifying the base system to adapt it to particular use cases. Most of the modifications are written in the implementation language of the VM, which has a direct access to the object representation and the execution environment. We illustrate the impact of these modifications on small programs written in the source language. The last modification example is written in the source language to show that it could be possible to change the semantics of the system without having to modify the implementation. Figure 4.1 illustrates conceptually the two possible channels of modification. Since it can be difficult to differentiate between the implementation language and the source language, a comment line at the beginning of each example explicitly states to which case they belong.

Note that in the current version of the system, an application that is aware it is running on Photon can modify the semantics of some operations, there is therefore no distinction between a modification written in the source language and an application that would use the features of Photon. Note also that instrumentations written in the implementation language are not interpreted or validated by Photon. They simply use abstractions provided by the system in the form of proxy objects and methods.

In both cases, the examples testify that the chosen abstractions encapsulate implementation details

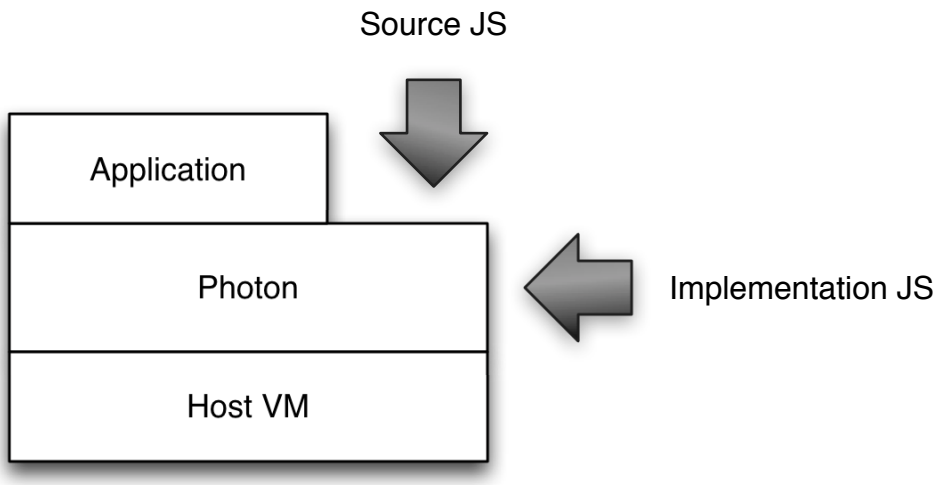

Figure 4.1: Modification channels

required for performance. The most common use case anticipated is for a researcher or developer to write an instrumentation or modification in the implementation language using the abstractions provided for gathering data about a running program. Once the instrumentation has been shared as part of a library, an end-user could load Photon in a web browser extension and activate and deactivate the instrumentation, while the program is running.

We first explore how to obtain object model operation information by redefining their corresponding methods. We then see how to obtain a dynamic call graph by redefining the `call` method on functions. Both of these support our thesis, by showing how the object model and the function calling protocol can be instrumented. Finally, we discuss ways to enforce runtime invariants used to catch common programming mistakes to show that although our focus was on building instrumentation infrastructure, the resulting system has broader applicability than the context in which it was developed.

4.1 Obtaining object model operation information

Given a good approximation of the performance cost of JS operations, such as property accesses and object creation, we might be interested in estimating the performance of an application by computing the number of run-time occurrences of each of these operations.

The design of the system makes it easy to do by wrapping the method implementing the semantic operation with a function incrementing a counter. In the following example, we do it for property accesses, property assignments and property deletions. The following example shows an example of instrumentation code for the object model, expressed in the implementation language:

```
// Language: Implementation JS
(function () {
    var results = {
        __get__:0,
        __set__:0,
        __delete__:0
    };

    var getFn = root.object.get("__get__");
    var setFn = root.object.get("__set__");
    var deleteFn = root.object.get("__delete__");

    root.global.set("startInstrumentation", clos(function ($this, $closure) {
        root.object.set("__get__", clos(function ($this, $closure, name) {
            results.__get__++;
            return getFn.call($this, name);
        }));

        root.object.set("__set__", clos(function ($this, $closure, name, value) {
            results.__set__++;
            return setFn.call($this, name, value);
        }));

        root.object.set("__delete__", clos(function ($this, $closure, name) {
            results.__delete__++;
            return deleteFn.call($this, name);
        }));
    }));

    root.global.set("stopInstrumentation", clos(function ($this, $closure) {
        root.object.set("__get__", getFn);
        root.object.set("__set__", setFn);
        root.object.set("__delete__", deleteFn);
    }));

    root.global.set("getInstrumentationResults", clos(function ($this, $closure) {
        var obj = root.object.create();
        obj.set("getNb", results.__get__);
        obj.set("setNb", results.__set__);
        obj.set("deleteNb", results.__delete__);
        obj.set("totalNb", results.__get__ + results.__set__ + results.__delete__);
        return obj;
    }));
})();
```

This instrumentation can be exercised with the following example, written in the source language:

```
// Language: Source JS
(function () {
    var o = {};

    o.foo = 2;
    o.foo;
    delete o.foo;

    startInstrumentation();
    o.foo = 2;
    o.foo; o.foo;
    delete o.foo; delete o.foo; delete o.foo;
    stopInstrumentation();

    o.foo = 2;
    o.foo;
    delete o.foo;

    var results = getInstrumentationResults();
    for (var p in results) {
        print(p + ": " + results[p]);
    }
})();
```

It should give a count of one for property assignments, a count of two for property accesses and a count of three for property deletions.

Notice that the instrumentation is granular in time, namely it can be activated and interrupted at any moment. To cover the entire application execution, instrumentation need simply be activated before an application is actually started.

4.2 Obtaining a dynamic call graph

A dynamic call graph is a data structure encoding the calling relationship between functions, occurring during the execution of a program. For example, if a function `a` calls a function `b`, the *a calls b* relationship will be registered in some way.

Dynamic call graphs can be used to determine the code coverage of unit tests as well as to provide detailed runtime information for code comprehension, run-time optimizations and analyses of source code.

Context-insensitive call graphs do not encode the calling context of a call, while context-sensitive call graphs do. For context-insensitive call graphs, we might represent functions as nodes and calling relationships as edges. Context-sensitive call graphs could represent functions as multiple nodes, depending on their respective callers. For the sake of simplicity, we will consider a context-insensitive call graph for the remainder of this example.

As seen previously in the design chapter (3), function calls origin from four sources in the language: global function calls, method calls, indirect calls through `call` and `apply` and direct calls to functions stored in variables. By redefining the `call` and `apply` methods, we can intercept every call made from

these four origins.

The following program will be traced and exhibits every case enumerated above. Four different functions are declared and two are initially called:

```
// Language: Source JS
// Global functions
function a()  { };           // Does not call any other
function b()  { a.call(); }; // Calling through "call"
function c()  { b(); };      // Calling a global function
function d()  { };           // Unused function

(function () {
    // Lexically scoped objects and functions

    // Creating an object with a single method
    var o = {m:c};

    // Direct recursive call closed-over itself
    function e(n) { if (n > 0) e(n-1); };

    // Method call
    o.m();

    // Direct call
    e(10);

    // Not called: d
})();
```

In this example, `c` calls `b`, `b` calls `a`, and `a` calls no other function, `e` recursively calls itself but no other function and `d` is not called nor calls another function. We would expect a call graph like the one shown in Figure 4.2 to model those relationships.

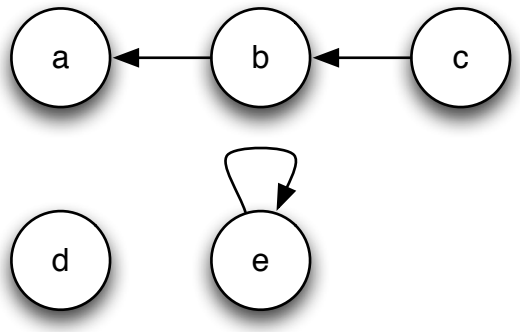

Figure 4.2: Call graph for example code

A context-insensitive analysis requires a single node per function. Therefore, we can store the profiling information directly on the functions themselves. This way it avoids the complexity associated with

maintaining separate data structures and maintaining a correspondence between functions and those data structures.

4.2.1 Protecting the profiling code in a critical section

A *critical section* usually refers to a section of code in a multi-threaded program in which the system guarantees that no preemption will occur. By analogy, we use the term to refer to a section of code in a profiled program in which the system guarantees that no profiling occurs.

By using a critical section, we solve the issue of having the profiling code being profiled. It is important to avoid profiling the profiling code, not only because it pollutes results but mostly because it can introduce *meta-stability* issues, namely that performing an operation in the profiling code might call back into the same operation being profiled, creating an infinite loop.

A critical section can be implemented by introducing a flag that controls whether profiling occurs or not. Just before the profiling code starts executing, the flag is set and right after the profiling code ends the flag is reset. It can be done with a boolean variable if the section needs not be reentrant or with a integer otherwise.

When profiling JS function calls, we need to be cautious about the different ways the flow of control can be manipulated by the callee. In JS, a function can return either normally through its explicit or implicit return statement or by raising an exception. Fortunately, JS provides a **`finally`** construct that guarantees that however the flow of control leaves a try block, the **`finally`** block will always be executed. We can therefore perform pre-call operations in a **`try`** block and perform post-call operations in a corresponding **`finally`** block.

The next requirement is to be able to intercept every function call in the program. Our design makes it easy by reducing every occurrence of function calls to sending the message `call` or `apply` to the called function, whether it is global, local or is a method. We can further implement `call` in terms of `apply`, which gives us a single point of instrumentation for the whole system. By redefining the `call` and `apply` methods on the root function, we can profile the whole system.

The only problem left with this approach comes from the circularity introduced by having every JS object operation and function invocation performing a `call`. It means that no primitive operation in the source language can be used to perform a call. We solve it by performing the instrumentation in the implementation language. [1]

The next example introduces two instrumentation primitives as global functions in the source language, `startCallInstrumentation` and `stopCallInstrumentation`. When called, the instrumentation function initializer redefines the `call` and `apply` methods on the root function and uses the `before` and

[1] Extensions to the source language could also have been made, in which case the source language could have been used to the same effect.

`after` source-language functions to perform pre- and post-call operations. By virtue of the critical section, any operation performed by `before` and `after` will not be profiled:

```
// Language: Implementation JS
(function () {
    var flag = 0;
    var defaultCallFn  = root.function.get("call");
    var defaultApplyFn = root.function.get("apply");

    root.global.set(
        "startCallInstrumentation",
        clos(function ($this, $closure, before, after) {
            var applyFn = clos(function($this, $closure, rcv, args) {
                if (flag !== 0) {
                    return $this.call.apply($this, [rcv].concat(args.unbox()));
                }

                try {
                    flag++;
                    before.call(null, $this);
                    flag--;
                    var r = $this.call.apply($this, [rcv].concat(args.unbox()));
                } finally {
                    flag++;
                    after.call(null, $this);
                    flag--;
                }
                return r;
            });

            root.function.set("call", clos(function($this, $closure) {
                return applyFn.call(
                    $this,
                    arguments[2],
                    new ArrayProxy(Array.prototype.slice.call(arguments, 3))
                );
            }));
            root.function.set("apply", applyFn);
        })
    );

    root.global.set(
        "stopCallInstrumentation",
        clos(function ($this, $closure) {
            root.function.set("call", defaultCallFn);
            root.function.set("apply", defaultApplyFn);
        })
    );
})();
```

This definition has an interesting property: the initialization of the call instrumentation is atomic and will take effect the next time a call is performed but will not affect any function currently in the calling context. Likewise, the instrumentation can be changed or stopped on-the-fly but the previous instrumentation will still performs its post-call operations on functions currently in the calling context.

The previous listing was essentially an extension of the implementation to allow function call instru-

mentation. The next listing uses that extension to build a dynamic call graph. It registers all called functions on a shadow stack as well as the calling relationship between the function on top of the shadow stack and the called function. A predicate is used to avoid registering calls to functions that do not have an identifier (`__id__` property). [2] The `dynCallGraphResults` function prints the call graph in the DOT language [3] for visualization. This instrumentation can be expressed in the source language:

```
// Language: Source JS
var callgraph = {};
var callstack = ["global"];

function beforeApply(fn) {
    if (fn.__id__ !== undefined) {
        var caller = callstack.length - 1;
        var node = callgraph[callstack[caller]];
        if (node === undefined) {
            node = {};
            callgraph[callstack[caller]] = node;
        }
        callstack.push(fn.__id__);
    }
}

function afterApply(fn) {
    if (fn.__id__ !== undefined) {
        callstack.pop();
        var node = callgraph[callstack[callstack.length - 1]];
        node[fn.__id__] = true;
    }
}

function dynCallGraphResults() {
    var str = "digraph {";
    for (var caller in callgraph) {
        if (callgraph.hasOwnProperty(caller)) {
            var set = callgraph[caller];
            var callee = "";
            for (var callee in set) {
                if (set.hasOwnProperty(callee)) {
                    str += caller + " -> " + callee + ";";
                }
            }
        }
    }
    str += "}";

    print(str);
}

startCallInstrumentation(beforeApply, afterApply);
```

[2] Identifying functions in JavaScript is non-trivial because some of them are anonymous, they might be stored in more than one variable and their variable name might conflicts with other locally-defined variable names elsewhere in the program. For this particular example, an extension to the source language was made that uses the global variable name as an identifier. A different naming strategy might be used without impact on the code presented.

The instrumentation performed will output the expected call graph as shown in the previous section. Running both the extension and the instrumentation on the example code given at the beginning of the section will output the correct dynamic call graph.

4.3 Enforcing run-time invariants

Instead of extending the behavior of operations to perform instrumentation, the next examples *modify* the semantics of the original operation to provide a different behavior, with the goal of enforcing run-time invariants. This goes beyond our thesis to provide a glimpse of the usefulness of the system beyond instrumentation tasks.

4.3.1 Ensure that all accesses are made to existing properties

The default semantics of property access in JS specifies that accessing a non-existing property should return `undefined`. It has the unfortunate consequence that the presence or absence of a property on an object is ambiguous: if an existing property has `undefined` as a value, we cannot tell if the property is present or not by the return value of the property access operation. Combined with the automatic conversion of values for most operations, an unintended missing property on an object might cause obscure bugs to crop up later in the program.

We can provide a *fail early* semantics to the property access operation by raising an exception if a property is not present. This can be done easily by replacing the `__get__` operation on the root object. The following example uses the implementation language to have access to the object representation methods directly:

```
// Language: Implementation JS
(function () {
    print("Retrieving original operation");
    var get = root.object.get("__get__");

    print("Replacing the semantics of the operation");
    root.object.set("__get__", clos(function ($this, $closure, name) {
        var obj = $this;

        while (obj !== null) {
            if (obj.has(name))
                return obj.get(name);

            obj = obj.getPrototype();
        }

        throw new Error("ReferenceError: property '" + name + "' not found");
    }));

    print("Testing the semantics of the operation");
    try {
        send(root.object.create(), "__get__", "bar");
        print("Should not be reached");
    } catch (e) {
        print(e);
    }

    print("Restoring the original behavior");
    root.object.set("__get__", get);
    print(send(root.object.create(), "__get__", "bar"));
})();
```

Having separate cases for the presence of a property with an *undefined* value and an absent property, an exception can be thrown only in the second case. A similar method could be used to check arrays for out-of-bounds accesses or deletions that could migrate the representation from dense to sparse.

4.3.2 Ensure that a constructor always returns the object it receives

The semantics of object construction in JS can be surprising. When creating an object with a call to a function with the **new** operator, a new empty object is created behind the scene and the function will be called with a **this** value bound to the created object. When the function returns, the return value type is inspected. If it is an object, the **new** expression will yield the returned object. However, if it is a primitive value, the **new** expression will yield the object created before the call to the function.

This behavior guarantees that an object will always be returned and allows constructors to be written more succinctly by omitting the **return** statement. However, it might also be a source of run-time errors if a constructor is later modified and an object is unintentionally returned.

We can dynamically change the semantics of the construction operation to throw an error if an object different than the original object created is returned by the constructor. For this last example, the modification is done in the source language to show that by exposing the semantics of operations

as methods, some modifications can be done without having to think in terms of the implementation language:

```
// Language: Source JS
(function () {
    print("Remembering the old behavior");
    var ctor = Function.prototype.__ctor__;

    print("Replacing the constructor behavior");
    Function.prototype.__ctor__ = function () {
        var o = Object.create(this.prototype);
        var r = this.apply(o, arguments);

        if (r !== o)
            throw Error("--> Constructor did not return the 'this' object");

        return o;
    }

    print("Creating a new faulty constructor");
    function Foo()
    {
        this.foo = 42;
        // implicitly returns the undefined value
    }

    print("Calling the faulty constructor");
    try
    {
        new Foo();
    } catch(e)
    {
        print(e);
    }

    print("Restoring the old behavior");
    Function.prototype.__ctor__ = ctor;

    print("Testing the old behavior");
    new Foo();
})();
```

The two previous examples show the flexibility of our implementation beyond mere instrumentation tasks. The fact they could both be written with a minimal amount of knowledge of the implementation brings the possibilities of modifying the language semantics to non-implementers.

Now that we have seen what we gained from a flexible implementation, we will investigate the performance of the resulting system.

Chapter 5

Performance

"There is a deep difference between what we do and what mathematicians do. The "abstractions" we manipulate are not, in point of fact, abstract. They are backed by real pieces of code, running on real machines, consuming real energy and taking up real space. To attempt to completely ignore the underlying implementation is like trying to completely ignore the laws of physics; it may be tempting but it won't get us very far."

— Gregor Kiczales [14]

The dynamic instrumentation of an application inevitably introduces a performance cost. The observed performance slowdown of an instrumented application compared to an uninstrumented application can be attributed to two sources: the *inherent overhead* introduced by the instrumentation infrastructure, in our case Photon, and the *instrumentation overhead*, specific to the data being gathered and the processing of that data while the application is running.

Photon has been designed to be used for the instrumentation of web applications. These applications usually exhibit a mix of JavaScript operations, Document Object Model manipulations, for rendering the graphical part of the application, and communication exchanges with servers over the network, to obtain and send information. The most recent evaluations of the behavior of web applications, in 2009 [25] and 2010 [28], showed that these applications were mostly event-driven with functions executing for a few milliseconds at a time, therefore they were not bound by the peak performance of JS VMs, optimized for CPU-bound and batch-oriented tasks. We therefore anticipate that Photon would introduce a small perceived overhead on real-world applications, since an overhead is only introduced on a subset of JavaScript operations while the rendering operations of the browser and the network communications are unaffected.

For evaluating the performance of Photon, we chose CPU-bound, JavaScript-only tasks, to evaluate the worst-case schenario. It allowed us to perform the evaluation independently of a full integration with a web browser. We use the V8 benchmarks version 7 because they stress many features of the language,

some that Photon reifies, such as object operations and some that Photon does not, such as control-flow operations and regular expression matching.

Photon aims to simplify the task of writing dynamic instrumentations, that previously were made by modifying the source code of a production interpreter. We therefore compare the inherent overhead of Photon to the performance of a production interpreter with its JIT-compiler disabled. The major finding of this performance evaluation is that the overall inherent overhead of Photon is sufficiently low that Photon executing over a VM with a JIT-compiler is faster than the Firefox interpreter. It means that VM layering can be a competitive approach compared to the modification of a state-of-the-art interpreter.

In the next sections, we first evaluate the overhead of the approach and relate it to the nature of the benchmarks. We show the necessity of optmizing the message sends. We then compare the inherent overhead when running over different JITs to the bare execution of benchmarks over the Firefox interpreter. Finally, we measure the instrumentation overhead of a simple instrumentation.

5.1 Setting

We used the latest versions, as of March 2013, of three of the major browsers, Safari, Chrome and Firefox.

- **Safari** version 6.0.2 (8536.26.17), which is based on the Nitro JS VM.
- **Chrome** version 25.0.1364.172, which is based on the V8 JS VM.
- **Firefox** version 20.0, which is based on the SpiderMonkey JS VM. Firefox was run with the JIT enabled, and also with the JIT disabled (which causes the SpiderMonkey interpreter to be used). To disable the JIT we have used the following Firefox settings which were suggested by the SpiderMonkey development team:

  ```
  javascript.options.ion.content        false
  javascript.options.methodjit.chrome   false
  javascript.options.methodjit.content  false
  javascript.options.typeinference      false
  ```

  Note that disabling SpiderMonkey's type inference actually accelerates the execution of all programs because the interpreter does not take advantage of the type information.

To simplify the description of the results, we conflate the name of the web browser with that of its JS VM.

A computer with a 2.6 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM and running OS X 10.8.2 is used in all the experiments. The table results are provided in Appendix A. Unless otherwise indicated, the values reported in the tables are the averages over five executions.

5.2 Performance with no instrumentation

Figure 5.1 shows the inherent overhead of Photon in terms of its slowdown factor, a multiple of the execution time required to perform the same task. The slowdown factor is illustrated on a logarithmic scale. The results for each benchmarks are grouped for each of the VMs evaluated. Minimal and maximal values are shown near their corresponding benchmarks and the goemetric mean is provided to compare VMs.

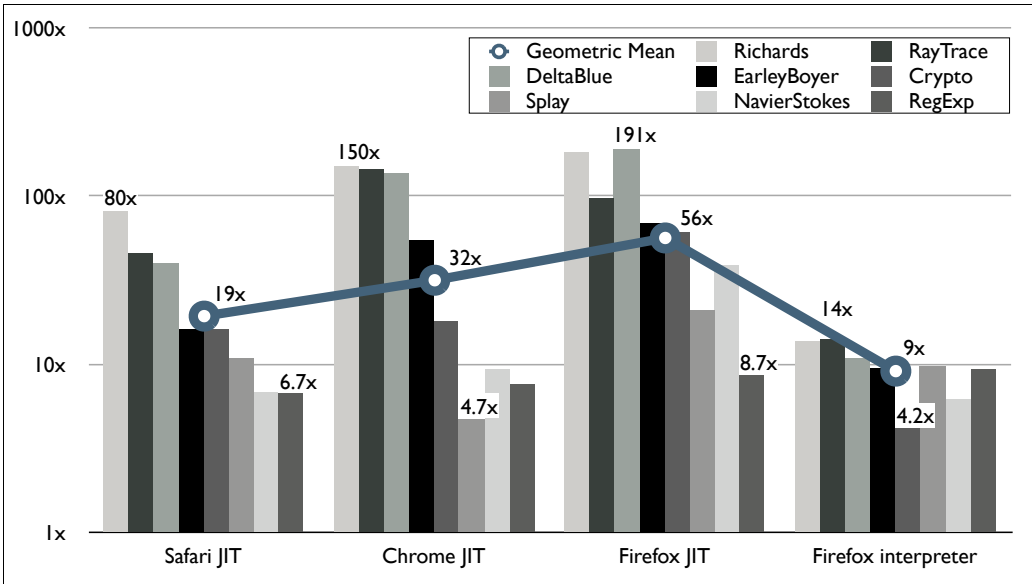

Figure 5.1: Inherent overhead of Photon on the V8 benchmark suite on each JS VM. For each VM, the minimal and maximal slowdown factors are shown near their corresponding benchmark.

Notably, the slowdown incurred on the Firefox interpreter is both lower, by a factor of 2 to 6 times, and shows less variability, with a range of ≈10× rather than ≈70× to ≈180×, than the slowdowns observed on other JIT-based VMs. We attribute these results to the aggressive optimizations performed by JIT VMs, which are defeated by the nature of the code transformation Photon performs. We performed another experiment, whose results are not shown, where all function bodies were enclosed in a **`try finally`** statement, knowing that current compilers do not optimize **`try`** bodies. We then observed that the performance of the JIT-based VMs became similar to that of the Firefox interpreter.

For JIT-based VMs, the important variablity of the results accross different benchmarks illustrate partially the effect of the differential implementation strategy for Photon (some operations of the language are reified while other are simply directly reused). The effect is clear on the RegExp benchmark where

a significant part of the time is spent executing the matching algorithm of regular expressions, which Photon directly reuse from the underlying VM.

For other benchmarks, the effect is less evident and the higher slowdowns can be attributed to the ability of the JIT compiler to optimize the reified operations. Figure 5.3 gives a rough approximation of the nature of JavaScript operations performed by each benchmark. Each pie slice area is proportional to the ratio of the number of each of these operation performed during execution to the total number of operations instrumented. For example, almost a quarter of all operations counted for the Richards benchmark are accesses to the **`this`** object, from within a method. Although, NavierStokes and RegExp have a similar proportion of object accesses (`ObjectGetExpression`) compared to Richards, RayTrace and DeltaBlue, a fifth compared to a fourth or a thirth, their execution time is significantly faster, by at least an order of magnitude. It suggests some operations such as array accesses are much better optimized than other object accesses.

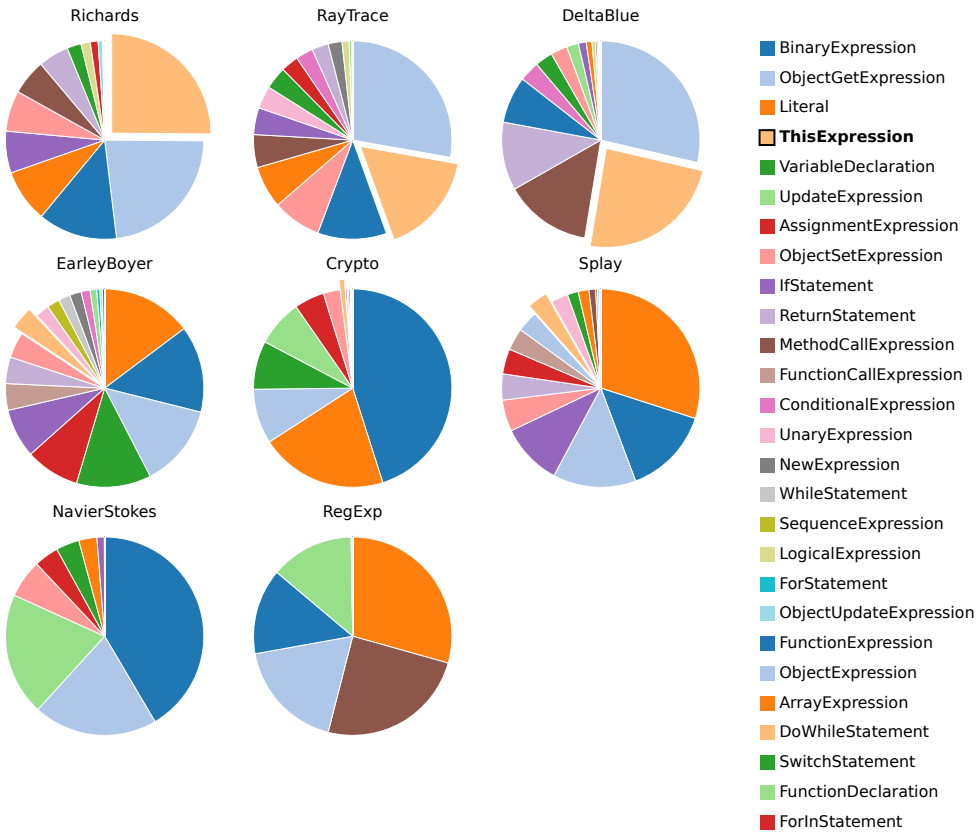

Figure 5.2: Comparison of the ratios of operations for the V8 benchmarks.

Comparing the frequency of each operations to the observed slowdown yielded a confirmation of the

validity of the differential strategy and an interesting insight. For all control-flow operations, no clear correlation could be found, which is unsurprising since the operations are not reified. However, of all the operations observed, it is the frequency of the **`this`** operation that seems to correlate the most to the slowdowns observed. Figure 5.3 shows again the measured slowdowns, grouped by benchmark and on a linear scale, and compares it to the ratio of **`this`** operations as in Figure 5.2. We attribute thie correlation to the impact of the modification of the calling convention of functions, in which the **`this`** object is explicitly passed as an argument instead of implicitly, as would be the case in the original JavaScript code. It is likely that the compiler can optimize away type tests on operations using the **`this`** identifier within a method because it cannot be assigned, therefore the object refered to remains constant throughout the method. This can in turn facilitates, inlining of method calls using **`this`** as the receiver object.

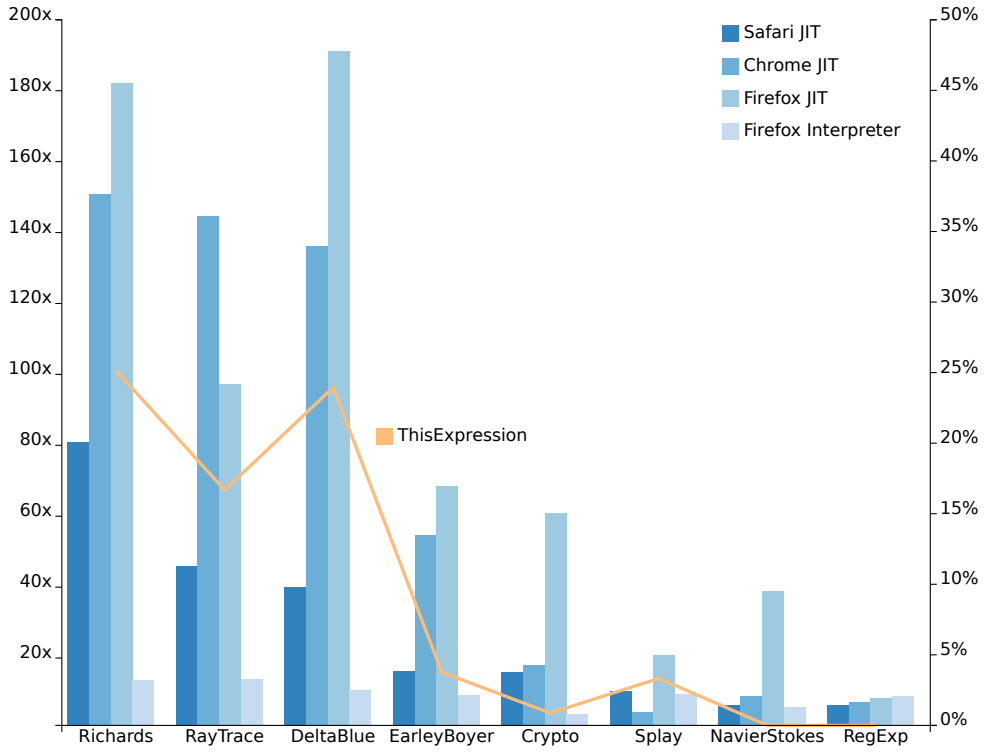

Figure 5.3: Comparison of the inherent overhead of Photon with the measured ratio of **`this`** operations during execution.

5.3 Effect of send caching

Figure 5.4 illustrates the effect of the caching strategy introduced in Section 3.1.3 by showing the *additional* slowdown incurred by disabling the optimization. These figures should be *multiplied* by the slowdowns shown in Figure 5.1 to obtain the slowdowns compared to the execution of the benchmarks without Photon. The EarleyBoyer benchmark is not shown because its execution requires a stack size bigger than the amount of stack space provided by the VMs.

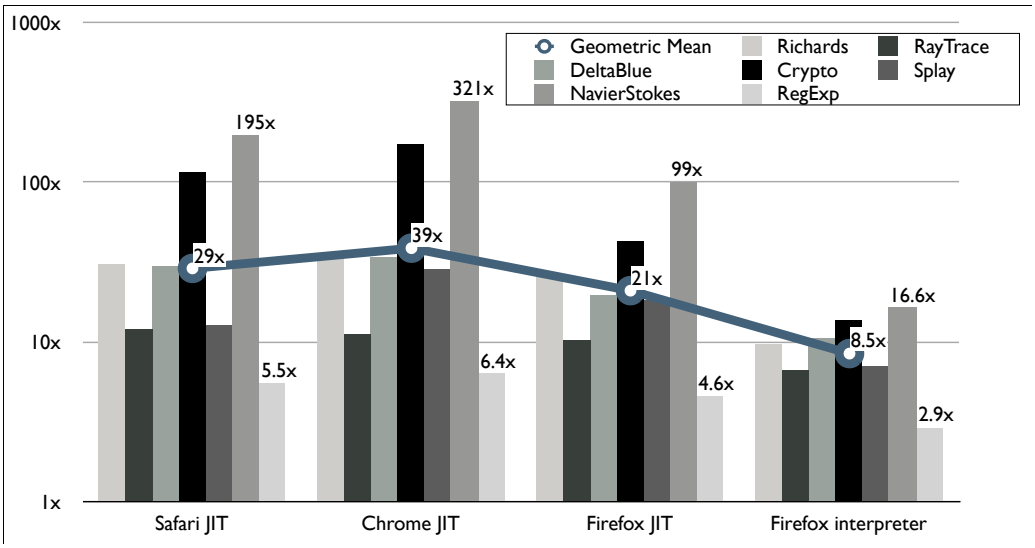

Figure 5.4: Execution speed slowdown of Photon on the V8 benchmark suite on each JS VM when send caches are deactivated.

Notice that the RegExp benchmark is the least affected, again because most of the time is spent in the regular expression algorithms, unaffected by Photon's transformation. The most affected benchmarks are NavierStokes and Crypto, both performing a high number of array operations.

Needless to say, with a maximum additional slowdown of 320x and geometric means between 8x and 40x, the caching optimization is critical to obtain acceptable performance number.

5.4 Comparison with interpreter instrumentation

The previous figures have shown the performance impact of Photon compared to a bare execution on the same VM. However, a more interesting comparison is made to the bare execution over an *interpreter*, since Photon is intended to replace instrumentation approaches that modify the implementation of a state-of-the-art interpreter.

Figure 5.5 shows the relative speed of the benchmarks executing on Photon over the different JIT-based VMs compared to the execution on the Firefox interpreter. A bar smaller than one means that Photon is faster than the interpreter. Safari and Chrome JITs perform almost equally well, with all but the RayTrace benchmark being within a factor of two and an overall speed faster than the Firefox interpreter. The Firefox JIT however is more than two times slower. Notice how the NavierStokes and Crypto benchmarks are more than seven times *faster* than their interpreter counterpart, while performing object operations and binary operations on numbers and strings. This is were the layered VM strategy seems to really shines compared to the instrumentation of an interpreter, by allowing the JIT to optimize through the additional abstraction layer.

The small performance difference between the Safari and Chrome JIT, but significant in the worst case on the RayTrace benchmark, suggests prefering the former if the browser implementation does not matter for a particular instrumentation.

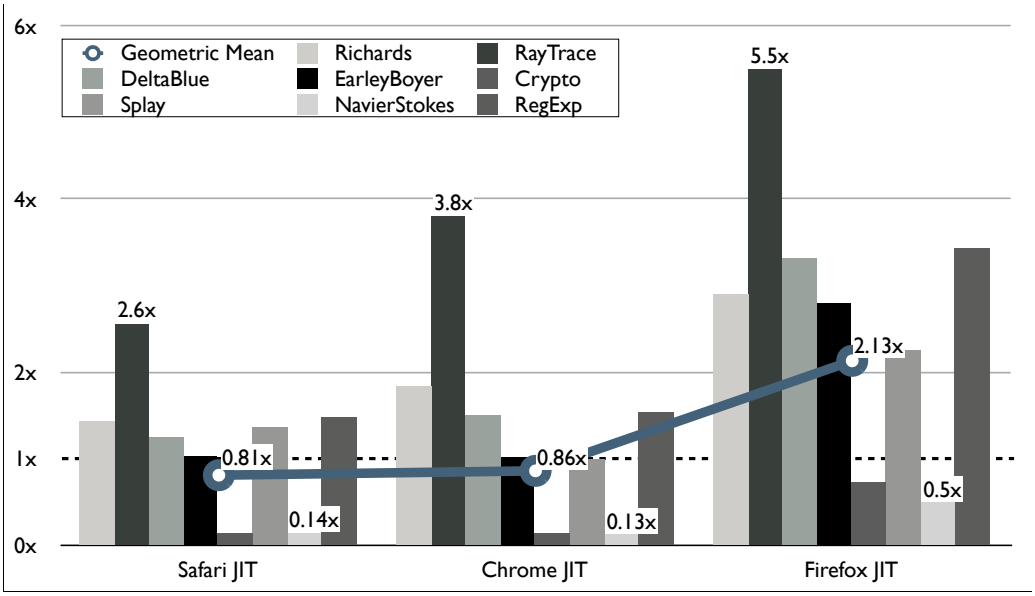

Figure 5.5: Performance of the V8 benchmark suite executed by Photon without instrumentation on each JIT VM compared to the benchmark running directly on the Firefox interpreter. The height of the bars indicates the execution speed ratio (smaller than one is better for Photon).

5.5 Performance with instrumentation

Finally, we have evaluated the performance of Photon with an instrumentation that counts the number of run-time occurrences of the following object representation operations: property read, write and deletion. We chose this particular instrumentation because it is simple, it covers frequently used object model operations and it was actually used to gather information about JS (it can be used to reproduce the object read, write and deletion proportion figure from [28]).

Two implementations of this instrumentation were used; a simple and a fast version. The simple version does not exploit memoization and corresponds to the straightforward implementation: incrementing a counter and calling the corresponding object representation operation. The fast version uses the memoization protocol to inline the counter incrementations inside the optimized version of the object operations.

The simple version is intended to measure the performance that can be expected from a quickly developed instrumentation while the fast version is intended to measure the performance impact of the instrumentation operations alone. This is therefore a low-barrier high-ceiling example and illustrates the flexibility that can be gained when the choice of aiming for performance is left to users of the system. Not counting the result formatting code, the simple version is 16 lines of JS code and the fast version is 100 lines of JS code.

The execution speed slowdown of Photon on Safari is given in Figure 5.6. The slowdowns for the simple version are between 1× and 29× but are less than a factor of 2× for all the benchmarks with most benchmarks close to 1×. This means that on Safari JIT, on average, the benchmarks run with the fast version of the instrumentation on Photon essentially at the same speed as the uninstrumented original benchmarks directly on the Firefox interpreter.

Results for other VMs are given given in Tables A.4 and A.5, showing that the results are similar for the Chrome JIT.

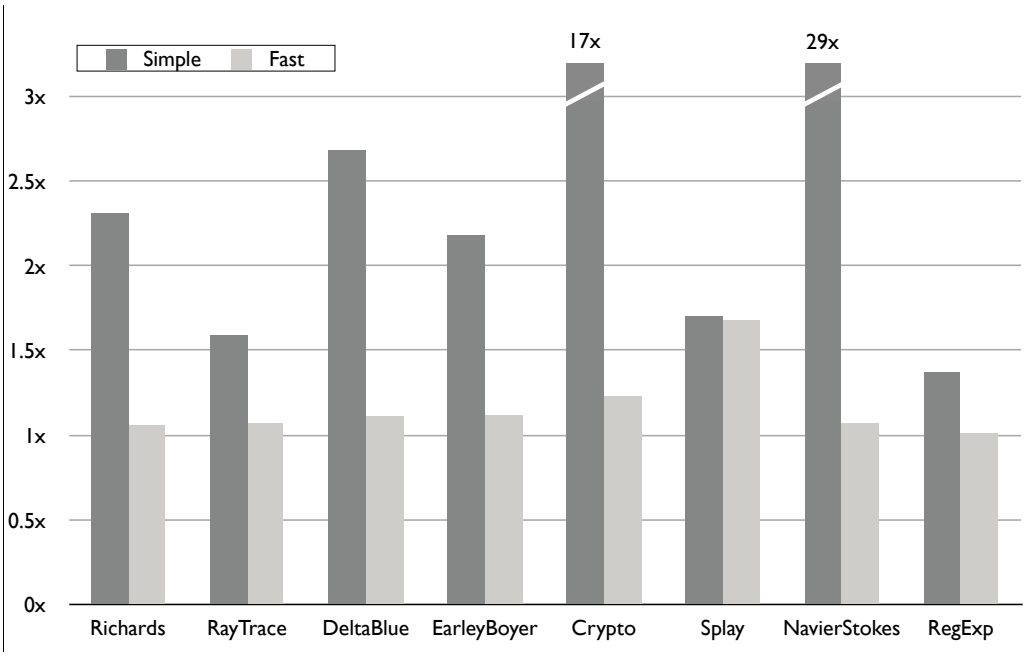

Figure 5.6: Execution speed slowdown of Photon with a simple and a fast instrumentation of property read, write and delete on the Safari VM.

Chapter 6

Conclusion

"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."

— Winston Churchill

The preceding chapters explained how a metacircular VM targeting its source language, based on a message-sending object model could provide flexible run-time instrumentation of the object model and function-calling protocol with an inherent overhead within the performance of a state-of-the-art interpreter, without having to modify the underlying VM source code. The motivation for the work stems from the effort involved with the current approach, namely to manually instrument a production interpreter and maintain it up-to-date. This dissertation explained how to solve the core technical issue behind this state of affairs.

However, there is still some work to be performed to *actually* use those results to gather empirical data about JavaScript programs. At the same time, the approach suggests other possible uses which might be worth exploring. In the next sections, we discuss the limitations of the current prototype, with regard to its JavaScript implementation, and we identify how the approach could be broadly applied by improving upon our current results.

6.1 Limitations

The limitations of our current prototype come from JavaScript peculiarities that might be eliminated if the next versions of the standard were allowed to relax strict backward compatibility. Another option would be to perform run-time checks to ensure they never arise. However, we conjecture that the resulting system is more useful by relaxing the strict adherence to the behavior expected from current VMs for substantial performance gains. In the face of numerous quirks and warts in the design of the JavaScript

language, it is more important to provide a useful and powerful system, but potentially incorrect, than an irremediably slow but correct one.

6.1.1 Accessing the `__proto__` property leaks the internal representation

This limitation can affect the soundness of the program. This could have been solved at the design stage of JavaScript, if the access to the prototype of an object has been made through a method call, such as `getPrototype()` instead of by accessing the `__proto__` property. This is a problem of mixing meta-level with base-level information.

The problem can be mitigated with no run-time penalty by detecting, at compile-time, accesses to the `__proto__` property and calling the object representation `getProtype` method instead. However, the possibility of dynamically generating the `__proto__` name renders it unsound. It is yet to be seen if this actually happens in the wild.

6.1.2 Meta-methods can conflict with regular methods if they have the same name

This limitation will be solved in the next version of the standard, when unforgeable names will be available in user space. Until then, we can rely on seldom used names to minimize possible conflicts with existing code.

6.1.3 Setting the `__proto__` property throws an exception

This might be fixed by invalidating all caches should the prototype of an object change. A more sophisticated mechanism could be devised if the operation is frequent.

6.1.4 Operations on *`null`* or *`undefined`* might throw a different exception

When trying to access a property or call a method on the *`null`* or *`undefined`* value, VMs for JS throw an exception telling the user that the property or the method does not exist on the object. Our current design reifies property accesses as a call to the `get` method on the object representation and calls as a call to the `call` method, *once the object operation has been cached*. If the receiver is *`null`* or *`undefined`* the exception will tell of a missing method that is different from the property or the method called.

This problem actually only happens when instrumenting incorrect programs. It might not be a problem at all when browsing the web. Otherwise, it would be possible to test for *`null`* or *`undefined`* on every operations, at a substantial run-time cost.

Another option would be to change the exception being raised by adding test and patch code for that particular exception at every catch site, at a potentially more reasonable cost depending on the actual

usage of the catch construct.

This is fundamentally a problem of non-uniformity in the object model. It would not exist if every value was an object. This could be fixed in a subsequent revision of JavaScript by providing auto-boxing of `null` and `undefined`, in a similar fashion as what is done for other primitive types, and adding a "does not understand" handler for property accesses and method calls that could be overwritten. The default behavior could be to raise an exception when accessing, setting or deleting a property or calling a method. The prototype object for `undefined` and `null` could then have methods that raise proper exceptions.

6.1.5 Limited support for eval

Our compiler does not support accessing the local environment of a function from evaluated code. This can be fixed by maintaining the environment information on functions. This environment can then be provided to our compiler, when it is called during an `eval` operation.

6.1.6 Function calls implicitly made by the standard library are not intercepted

Functions passed to the standard library are wrapped to remove the extra arguments introduced by our compilation strategy. Should the need to intercept those calls arise, the wrappers could perform a message send instead of a direct call.

6.2 Future work

The obvious short term work would be to package the system in a browser extension for Firefox or Chrome and use it to try to replicate the results obtained on the dynamic behavior of JavaScript programs [28], to see if they still hold and try the instrumentation on more than just the 100 more popular websites. Then we could extend the instrumentation to cover not only the object model operations and function calls but also the binary, unary, control-flow and reflexive operations.

We believe there is much more potential to the current approach. The biggest open question is how close we can come to native performance while providing flexibility unavailable in the vanilla implementation. We compared ourselves to a state-of-the-art interpreter because this is what is being used for instrumentation nowadays. Similar performance is therefore enough for obtaining information about the dynamic behavior of JavaScript programs.

We argue that as virtual machines for dynamic languages become faster and incorporate more sophisticated optimization techniques, the implementation techniques for layered VMs should become worthy of research efforts since they have the potential to provide an efficient way of supporting numerous languages with limited efforts. This dissertation is a step in that direction.

Also, there could be qualitative improvements if the performance was much better, by opening whole new possibilities of applications. In the next sections, we reflect about possible approaches to improve efficiency using known techniques.

6.2.1 Improve compilation speed

Our OMeta-based compiler [36] can be significantly slow during the parsing stage. If compilation speed becomes critical, it could be replaced with a different algorithm or the OMeta runtime and compiled code could be optimized.

6.2.2 Allow a finer-grained redefinition of meta-methods

Object-model operations can only be redefined on the root object, mostly to simplify the tracking mechanism. A more sophisticated tracking mechanism could allow object model operations to be redefined for subsets of all objects by redefining the operation at the appropriate place on the inheritance hierarchy.

6.2.3 Efficient instrumentation

There are two major areas for optimizations: the inherent overhead of the system and the performance while the code is instrumented. We briefly touch upon each of them.

6.2.3.1 Improve the inherent overhead

The easiest known technique that would be worth trying to replicate in a metacircular setting is the dynamic recompilation of the source code based on type-feedback obtained during execution. It would further allow the specialization of the code to the actual types occurring during execution and the removal of message-sending caches for operations that have not been redefined.

Our current object representation makes it easy by wrapping the function being executed. The function could be replaced at run time with a specialized version. This could eliminate type tests and specialize functions used as constructors. Our current inline cache design allows the accumulation of arbitrary information which could be used for that purpose.

6.2.3.2 Provide mechanisms to optimize the instrumentation code

First, the system internals currently use a memoization protocol to specialize base operations. By exposing the mechanism to user code, this would allow analyses to specialize their behavior based on previous executions. This could notably be used to avoid performing operations in expensive data structures the second time.

Second, the system could offer a better granularity of instrumentation. Currently, instrumenting a base operation applies to all objects. However, if we could target only a subset of objects and handle

that subset at the cache level, we could avoid the instrumentation cost for a majority of objects. This could be done by implementing the equivalent of polymorphic inline caches and having a finer-grained tracking mechanism for cache states.

Bibliography

[1] Google Caja. `http://code.google.com/p/google-caja/`, December 2012.

[2] ADSafe. `http://www.adsafe.org/`, March 2013.

[3] DOT language. `http://www.graphviz.org/`, April 2013.

[4] ECMAScript 6. `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`, April 2013.

[5] JSProbes. `http://brrian.tumblr.com/search/jsprobes`, March 2013.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience*, 23(15):1749–1773, 2011.

[7] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM.

[8] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations*, pages 29–31. Winter Simulation Conference, 1968.

[9] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[11] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 111–122, New York, NY, USA, 2012. ACM.

[12] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor induction and meta-evaluation. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 153–168, New York, NY, USA, 1973. ACM.

[13] A. C. Kay. The early history of Smalltalk. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM.

[14] G. Kiczales. Towards a new model of abstraction in the engineering of software. In *Proceedings of the International Workshop on New Models for Software Architecture 1992; Reflection and Meta-Level Architecture*, pages 1–11, 1992.

[15] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[16] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript instrumentation in practice. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in javascript using aspects. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 361–376, New York, NY, USA, 2010. ACM.

[18] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.

[19] C. Maeda, A. Lee, G. Murphy, and G. Kiczales. Open implementation analysis and design. In *Proceedings of the 1997 symposium on Software reusability*, SSR '97, pages 44–52, New York, NY, USA, 1997. ACM.

[20] P. Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[21] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communication of the ACM*, 3(4):184–195, Apr. 1960.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[23] I. Piumarta and A. Warth. Self-sustaining systems. chapter Open, Extensible Object Models, pages 1–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[24] R. Rao. Implementational reflection in silica. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 251–267, London, UK, UK, 1991. Springer-Verlag.

[25] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, page 3. USENIX Association, 2010.

[26] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3), Sept. 2007.

[27] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of Javascript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM.

[28] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2010.

[29] M. Shaw and W. A. Wulf. Toward relaxing assumptions in languages and their implementations. *SIGPLAN Not.*, 15(3):45–61, Mar. 1980.

[30] B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[31] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 4–1–4–59, New York, NY, USA, 2007. ACM.

[32] B. Stroustrup. Bjarne stroustrup's FAQ. `http://www.stroustrup.com/bs_faq.html`, December 2012.

[33] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): sandboxing third-party scripts. In *Proceedings of the 3rd USENIX conference on Web Application Development*, WebApps'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[34] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 13–24, New York, NY, USA, 2010. ACM.

[35] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[36] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.

Appendix A

Performance results

	Safari JIT			Chrome JIT			Firefox JIT			Firefox interpreter		
	without	with		without	with		without	with		without	with	
Benchmark	Photon	Photon	Ratio	Photon	Photon	Ratio	Photon	Photon	Ratio	Photon	Photon	Ratio
Richards	11033	137	80.77×	16024	106	150.86×	12267	67	182.11×	195	14	13.71×
RayTrace	9866	214	46.06×	20889	144	144.66×	9725	100	97.34×	549	39	14.02×
DeltaBlue	7153	179	40.05×	20389	150	136.11×	12884	67	191.15×	224	21	10.88×
EarleyBoyer	10632	657	16.18×	36279	664	54.64×	16637	243	68.58×	679	71	9.53×
Crypto	19866	1233	16.11×	21073	1175	17.93×	14045	230	61.01×	168	40	4.15×
Splay	10506	973	10.80×	6252	1328	4.71×	12253	587	20.86×	1326	135	9.79×
NavierStokes	15511	2265	6.85×	24043	2562	9.39×	25519	654	39.00×	327	53	6.17×
RegExp	3813	568	6.71×	4151	545	7.62×	2128	245	8.70×	838	90	9.30×
Geom. mean	*10027*	*519*	*19.30×*	*15456*	*490*	*31.54×*	*11150*	*198*	*56.26×*	*421*	*46*	*9.09×*

Table A.1: Inherent overhead of Photon on the V8 benchmark suite on each JS VM. A measure of the execution speed, the *V8 score*, is given for the benchmark executed without and with Photon, as well as the ratio of the scores.

Benchmark	Safari JIT	Chrome JIT	Firefox JIT	Firefox interp.
Richards	30.42×	34.15×	25.61×	9.69×
RayTrace	11.90×	11.11×	10.15×	6.61×
DeltaBlue	29.57×	33.66×	19.65×	10.55×
Crypto	114.19×	171.28×	42.24×	13.57×
Splay	12.62×	28.49×	18.07×	7.02×
NavierStokes	195.26×	320.60×	98.85×	16.55×
RegExp	5.51×	6.39×	4.56×	2.88×
Geom. mean	*28.84*×	*38.60*×	*20.93*×	*8.45*×

Table A.2: Execution speed slowdown of Photon on the V8 benchmark suite on each JS VM when send caches are deactivated

Benchmark	Safari JIT	Chrome JIT	Firefox JIT
Richards	1.43×	1.84×	2.90×
RayTrace	2.56×	3.80×	5.50×
DeltaBlue	1.25×	1.50×	3.32×
EarleyBoyer	1.03×	1.02×	2.80×
Crypto	.14×	.14×	.73×
Splay	1.36×	1.00×	2.26×
NavierStokes	.14×	.13×	.50×
RegExp	1.48×	1.54×	3.43×
Geom. mean	*.81*×	*.86*×	*2.13*×

Table A.3: Performance of the V8 benchmark suite executed by Photon without instrumentation on each JIT VM compared to the benchmark running directly on the Firefox interpreter. The numbers indicate the execution speed ratio (smaller than one is better for Photon).

Benchmark	Safari JIT	Chrome JIT	Firefox JIT	Firefox interp.
Richards	2.31×	2.38×	2.81×	1.88×
RayTrace	1.59×	1.30×	2.19×	1.55×
DeltaBlue	2.68×	3.16×	2.03×	1.98×
EarleyBoyer	2.18×	2.31×	2.71×	1.78×
Crypto	16.80×	18.53×	6.91×	4.33×
Splay	1.70×	2.45×	1.96×	1.42×
NavierStokes	29.17×	39.41×	11.86×	5.65×
RegExp	1.37×	1.31×	1.29×	1.30×
Geom. mean	*3.54*×	*3.90*×	*3.03*×	*2.15*×

Table A.4: Execution speed slowdown of Photon with the simple instrumentation of property read, write and delete

Benchmark	Safari JIT	Chrome JIT	Firefox JIT	Firefox interp.
Richards	1.06×	1.26×	1.07×	1.24×
RayTrace	1.07×	.93×	1.02×	1.15×
DeltaBlue	1.11×	1.01×	1.02×	1.19×
EarleyBoyer	1.12×	1.14×	1.07×	1.15×
Crypto	1.23×	1.00×	1.00×	1.30×
Splay	1.68×	1.37×	1.05×	1.17×
NavierStokes	1.07×	2.05×	1.11×	1.36×
RegExp	1.01×	.99×	1.02×	1.03×
Geom. mean	*1.15*×	*1.18*×	*1.04*×	*1.19*×

Table A.5: Execution speed slowdown of Photon with the fast instrumentation of property read, write and delete