Université de Montréal

Harnessing performance for flexibility in instrumenting a virtual machine for JavaScript through metacircularity

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

Décembre, 2012

Résumé

Les développements récents sur les machines virtuelles (MVs), en particulier pour le language JavaScript, ont mis l'emphase sur la performance au détriment de la flexibilité. Cela limite notre compréhension du comportement dynamique des programmes en rendant laborieuse l'instrumentation de MVs existantes. Les approches existantes consistent à instrumenter manuellement un interpréte commercial, limitant l'acquisition de données longitudinales, à cause du coût élevé de maintenance de l'interpréte.

Ce mémoire montre que la flexibilité peut être récupérée dans l'instrumentation dynamique du modèle objet et des appels de fonctions, à un niveau de performance compétitif avec un interpréte commercial récent. Notre approche consiste à exécuter une MV méta-circulaire, ciblant le langage source, sur une MV rapide. Pour évaluer l'approche, nous proposons une MV pour JavaScript, nous présentons des exemples d'instrumentation et nous comparons la performance, avec et sans instrumentation, à l'interpréteur SpiderMonkey et la MV basée sur un JIT de V8.

Nous croyons que la combinaison de simplicité, de flexibilité et d'efficacité de notre MV est unique. Elle est rendue possible par trois contributions:

- L'unification des opérations réifiées du modèle objet et des appels de fonctions autour d'une primitive unique de passage de message, compatible avec la dernière version de JavaScript;
- Une implémentation efficace de la primitive de passage de message inspirée par la mémoisation *in situ*;
- Une représentation objet qui exploite les optimisations de mémoization *in situ* de la VM sous-jacente et le dynamisme du modèle objet pour obtenir des opérations virtualisées efficaces.

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

Abstract

Recent research and development on Virtual Machines (VMs), especially for the JavaScript language, has focused on performance, at the expense of flexibility. Notably, it has hindered our understanding of the run-time behavior of programs by making instrumentation of existing VMs laborious. Past approaches required manual instrumentation of production interpreters, preventing the acquisition of longitudinal data because of the high cost of maintaining the interpreter up-to-date.

This dissertation shows that performance can be harnessed to provide flexible run-time instrumentation of the object model and function-calling protocol at a performance competitive with a state-of-the-art interpreter, without having to modify the VM source code. Our approach consists in running a metacircular VM targeting the source language, based on a message-sending object model, on top of another fast VM. To demonstrate the approach, we provide a reference VM for JavaScript, we show the possibility of instrumenting the object model operations and function calls and we finally compare the performance with and without instrumentation to the SpiderMonkey interpreter and V8 JIT-compiler based VM.

We believe our combination of simplicity, flexibility and efficiency is unique. As such, this dissertation contains three original contributions:

- The unification of the reified object model operations and function-calling protocol around a single message-sending primitive while preserving compatibility with the current version of JavaScript;
- An efficient implementation of the message-sending primitive inspired by inline cache optimizations;
- An object representation exploiting the underlying VM inline caches and dynamic object model to provide efficient virtualized operations.

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

Contents

Résumé **ii**

Abstract **iii**

Acronyms **iv**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Flexibility 2

1.2 Virtual machine 3

1.3 Metacircularity 4

1.4 JavaScript 5

1.5 Object model 6

1.6 Function-calling protocol 7

1.7 VM paradigm vs instrumentation 7

1.8 Outline 8

2 Previous Work **9**

2.1 Historical roots 10

2.1.1 Openness 10

2.1.2 Extensibility 11

2.1.3 Dynamism 12

2.1.4 Performance 13

2.2 Instrumenting JavaScript VMs 13

3 Design **15**

3.1 Reification choices 16

3.2 Message-sending foundation 17
3.2.1 Semantics 17
3.2.1.1 Basic 18
3.2.1.2 Augmented function-calling flexibility 19
3.2.1.3 Memoization protocol 19
3.2.2 Translating reified operations to message-sending 20
3.2.3 Efficient implementation 21
3.2.3.1 Cache states and transitions 23
3.3 Object representation 25
3.3.1 Basic representation 26
3.3.2 Fixed arguments number specialization 28
3.3.3 Type or value specialization 29
3.3.4 Note 29
3.4 Compilation example 30

4 Flexibility **34**
4.1 Obtaining object model operation information 35
4.2 Obtaining a dynamic call graph 36
4.2.1 Protecting the profiling code in a critical section 38
4.3 Enforcing run-time invariants 41
4.3.1 Ensure that all accesses are made to existing properties 42
4.3.2 Ensure that a constructor always returns the object it receives 43

5 Performance **45**
5.1 Methodology 45
5.2 Baseline performance 47
5.2.1 Running times 47
5.2.2 Memory usage 49
5.3 Instrumented performance 50
5.3.1 Running times 53
5.3.2 Memory usage 53
5.4 Interpretation 53

6 Conclusion **62**
6.1 Limitations 62
6.1.1 Accessing the `__proto__` property leaks the internal representation 63
6.1.2 Meta-methods can conflict with regular methods if they have the same name . . . 63

6.1.3 Setting the `__proto__` property throws an exception 63

6.1.4 Operations on *`null`* or *`undefined`* might throw a different exception 63

6.1.5 Limited support for eval 64

6.1.6 Function calls implicitly made by the standard library are not intercepted 64

6.2 Future work 64

6.2.1 Improve compilation speed 65

6.2.2 Allow a finer-grained redefinition of meta-methods 65

6.2.3 Efficient instrumentation 65

6.2.3.1 Improve the baseline performance 65

6.2.3.2 Provide mechanisms to optimize the instrumentation code 65

Bibliography **67**

List of Tables

3.1 Call types and their equivalent message sends . . . 21
3.2 Object model operations and their equivalent message sends . . . 22
3.3 Cache states . . . 23
3.4 Cache Events . . . 25
3.5 Cache Transition Conditions . . . 26
3.6 Object representation operations and their interface . . . 33

5.1 Baseline performance on V8 benchmarks . . . 48
5.2 Baseline performance on SunSpider benchmarks . . . 55
5.3 Baseline memory usage on V8 benchmarks . . . 56
5.4 Baseline memory usage on SunSpider benchmarks . . . 57
5.5 Instrumented performance on V8 benchmarks . . . 58
5.6 Instrumented performance on SunSpider benchmarks . . . 59
5.7 Instrumented memory usage on V8 benchmarks . . . 60
5.8 Instrumented memory usage on SunSpider benchmarks . . . 61

List of Figures

3.1 Cache States and Transitions 24
3.2 Basic object representation 27
3.3 Special object representation 28

4.1 Call Graph for example code 37

Chapter 1

Introduction

"Impermanent are all component things,

They arise and cease, that is their nature:

They come into being and pass away,

Release from them is bliss supreme."

— Mahaa-Parinibbaana Sutta [28]

Computer systems are constantly modified to satisfy the evolving needs of their users. *Flexible* systems can evolve faster than rigid systems by placing fewer constraints on possible modifications, thus reducing the delay between the birth of an idea and its actual realization in a working system. Aiming for flexibility in the design of computer systems can make other properties, such as performance, security and reliability, easier to obtain by facilitating experimentation.

A *virtual machine* (VM) is a program that simulates the behavior of a computing machine. The simulated machine might exhibit properties that do not have physical equivalents. For example, by using a garbage collection component, it can provide the illusion of infinite allocatable memory. Programs inherit the properties and constraints of the VMs on which they run, therefore making them prime research objects to provide properties to an important number of programs.

Recent research and development on VMs, especially for the JavaScript language [12], has focused on performance, at the expense of flexibility. Notably, it has hindered our understanding of the run-time behavior of real-world programs by making instrumentation of existing VMs laborious and hard to maintain. As an example, three years ago some researchers manually instrumented a production interpreter to obtain execution traces [25]. At the time of writing this dissertation, the instrumentation work that was done cannot not be used because the interpreter that was modified is not part of the code base anymore.

Fortunately, the very same gains in performance can be used to regain flexibility, even on a rigid production VM. This dissertation shows that an efficient run-time optimizer can be harnessed to provide flexible run-time instrumentation of the object model and function-calling protocol at a performance competitive with a state-of-the-art interpreter, without having to modify the VM source code. Our approach consists in running a metacircular VM targeting the source language, based on a message-sending object model, on top of another fast VM. To demonstrate the approach, we provide a reference VM for an existing programming language, JavaScript, we show the possibility of instrumenting the object model operations and function calls and we finally compare the performance with and without instrumentation to the SpiderMonkey interpreter and V8 JIT-compiler based VM.

We believe our system, Photon, is unique in its combination of design choices that makes it simple, flexible and sufficiently efficient for data gathering of dynamic behavior for applications that can run acceptably fast on a modern interpreter. Previous systems for JavaScript targeting JavaScript as their runtime, such as Google Caja to enforce security invariants [1], Google Traceur to support the next version of JavaScript on existing VMs [2] and JSBench to record execution traces for automatic benchmark generation [24], use a source-to-source translation strategy as Photon do. However, the level of performance achieved by Photon, while isolating the application from the implementation of Photon is worth noticing. As such, this dissertation contains three original contributions:

- The unification of the reified object model operations and function-calling protocol around a single message-sending primitive while preserving compatibility with the current version of JavaScript;
- An efficient implementation of the message-sending primitive inspired by inline cache optimizations;
- An object representation exploiting the underlying VM inline caches and dynamic object model to provide efficient virtualized operations.

In addition to supporting the thesis above, we hope (1) to encourage future language designs to provide features facilitating an efficient layered approach to obtain flexibility and (2) that the object representation will be used by other language implementations seeking efficiency while targeting existing JS VMs.

1.1 Flexibility

Defining and evaluating flexibility is not an easy task. The usage of the term above appeals to the intuitive notion of a system easy to tailor to one's particular usage by modifying, removing or replacing its components. Easy might mean that there are few lines of code to modify to use a preconfigured option, that there are few manipulations to perform in a user interface or that unplanned extensions could be developed in a short time.

While intuitive, this definition is not sufficiently objective to compare different systems. It is tied to a subjective interpretation of efforts required to perform modifications. For the remainder of this dissertation, a more restricted definition of flexibility will be used. A system will be considered flexible if it exhibits the following four properties:

- *Open*: Its component behaviors can be modified by first-class data structures.
- *Extensible*: Its components can be independently modified or replaced and they support incremental definitions.
- *Dynamic*: Its components can be modified at run time.
- *Efficient*: The resulting system is fast enough for the task at hand and allows prompt feedback about the modification.

This definition serves both as a design goal for the resulting system and a way to compare different systems. The notions of *openness*, *extensibility*, *dynamism* and *performance* appear throughout this dissertation and are used to situate our work against the literature and other systems.

1.2 Virtual machine

A *virtual machine* is a program that simulates the behavior of a computing machine, that may or may not have a physical implementation. It may be identical to the physical machine on which it is executing, as is the case with current commercial solutions used to execute different operating systems as user processes, or it might be completely different, as is the case when executing a high-level language such as JavaScript.

In theory, there is no significant difference between a VM and an operating system. [1] They both act as a middle layer between the underlying machine and programs. They both provide abstractions and services common to all programs. In fact, VMs for high-level languages have been made to execute directly on hardware. In practice, the management of hardware peripherals, memory, processing units and network interfaces has been associated with operating systems and the support for higher-level features of programming languages has been associated with dedicated VMs. This historical distinction has allowed a plethora of languages to be available for application programmers by allowing language implementers to focus on supporting the semantics of programming languages instead of managing the physical machine resources.

The major drawback of simulating a computer is the important efficiency difference between a physical and a virtual implementation, the latter possibly being orders of magnitude slower than the former. Research on VM implementation has contributed techniques to minimize this efficiency gap. Various high-level languages, such as Java, JavaScript, Smalltalk, Scheme or Prolog, can now execute efficiently on general purpose processors making VMs practical for application development.

[1] "An operating system is a collection of things that don't fit into a language. There shouldn't be one." – Dan Ingalls [11]

In this dissertation, our focus is directed toward VMs constructed to support programming languages on top of existing VMs. The virtualization of operating systems, the management of physical resources as well as most of the low-level details required to execute directly on processors is ignored.

1.3 Metacircularity

VMs can be described by two languages:

- *Source language.* Language used by programs executing on the VM.
- *Implementation language.* Language that describe the behavior of the VM.

These languages can be different. For example, current commercial VMs for JavaScript (source language) are written in C++ (implementation language). A special case exists when the two languages are the same, with noteworthy properties.

When its source language and its implementation language are the same, a VM is said to be *metacircular*. Advantages of metacircularity are resource sharing and uniformity of the runtime. For example, a memory manager used to manage the internal data structures of a VM can be shared with the source language runtime. Uniformity allows optimizations written for the source language to also apply to the implementation language. In the context of an open system, source language code can replace implementation language code to modify the behavior of the VM. In the literature, *self-hosting* is also used to describe metacircular systems. We will restrict this latter term for describing a system that is *actually* used to produce new versions of itself. A metacircular VM needing special extensions still depends on an external machine to produce a different version of itself and is not self-hosting. Self-hosting is therefore a stricter definition. Self-hosting enables a faster evolution of the system by allowing functionalities developed for the source language to apply to the implementation language. It frees a system from the limitations of other available systems that would be needed otherwise. The VM presented in this dissertation is not self-hosting.

VMs can be implemented as interpreters when the behavior of programs is directly specified in the implementation language. However, VMs can also incorporate a compiler that performs translation of the programs to the language on which the VM is executing, while the program is executing. We say such a VM incorporates a Just-In-Time (JIT) Compiler.

We argue that a VM which incorporates a JIT compiler that targets its source language is possibly one of the simplest that can be built that mostly preserves performance characteristic of unmodified source elements. In fact, it amounts to a *differential* implementation, in which only the elements of interest in the language are acted upon. Optimizations for the new functionalities can target existing optimizations of the source language to efficiently implement them, without having to specify low-level mechanisms. There is even the possibility of faster-than-native performance if some features can be translated to equivalent features with better performance characteristics. For these reasons, a metacircular implementation that targets its source language was chosen.

A metric can be suggested to evaluate the cost of providing functionalities or properties that are not available in the original source language: the overhead of executing an original source program on the VM compared to bare execution on the reference machine. This metric will be used to evaluate the performance cost of the flexibility gained from the suggested design.

1.4 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). The next version of the standard is expected to add proper tail calls, rest parameters, block-scoped variables, modules, and many other features.

Initially, our research initiative chose JS as a research object because it is widely used to write web applications and there is a performance gap between benchmarks executing on the fastest implementations of JS and compiled with C/C++ compilers, suggesting there are techniques to be found to close the gap. During exploration of VM implementation techniques, the dynamic nature of the language was found to be particularly well-suited to base a flexible design around it and initiated the work presented here. Notably, the ability to use the underlying JIT compiler by way of `eval` and the `Function` constructor function also enables the *differential* implementation of a JIT compiler by adding preliminary phases to the compilation of the source programs.

1.5 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 will 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*.

1.6 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.

1.7 VM paradigm vs instrumentation

It can legitimately be asked what the difference is between a metacircular VM executing on top of an existing VM and a system that would perform source-to-source translation at run time. This difference is mostly conceptual and users of these systems need not know the difference. However, there seems to be a fundamental difference that governs the range of implementation techniques available to the implementer: what assumption is made about the possibility of source code inadvertently interfering with implementation code. In other words, does the system make an *open-world* assumption or a *closed-world* assumption.

In the first case, we will say that the system performs instrumentation of the source code. In the second case, we will say that the system can be considered a virtual machine, running on top of another virtual machine.

Our system makes a closed-world assumption to use the JavaScript global object for optimizations of implementation mechanisms.

1.8 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

Turning our attention to previous work can serve a number of purposes such as establishing the origin of fundamental concepts behind an idea, collecting learning material for the interested reader and demonstrating the originality of the work presented. In this chapter, all three will be addressed but this dissertation will emphasize the first two.

The terminology around flexible computer systems is both varied and vague. Synonyms to flexibility abound: *malleability* brings images of amorphous substrates, *versatility* testifies for the universality of the artifact, *plasticity* suggests a solid while reconfigurable medium and *customizability* reminds of color options for mass-produced consumers goods. While nuanced in their suggestive tones, usage of these terms share the unfortunate characteristic of lacking a precise definition which might serve as a basis for expectations toward the system they describe.

The consequence is that comparison between different systems built around different organizing principles is difficult and becomes almost impossible when the systems belong to different sub-disciplines, such as operating systems and virtual machine construction. Asserting that a system is flexible amounts to little more than a wish by the authors that a system will be easy for others or themselves to use for needs beyond the documented cases.

Untangling the issue of characterizing existing flexible systems according to the same framework is beyond the scope of this dissertation, since first and foremost, we are trying to show that performance can be used to gain flexibility. A taxonomy of existing systems would not help with the endeavor. However, this does not mean that we will fall prey to a "flexibility relativism". The four properties of *openness*, *extensibility*, *dynamism* and *performance* will guide our review of previous work, by grounding our understanding of flexibility in a richer and recognized definition. Still, the exact meaning of those terms vary in the literature but we shall be careful in precising our usage.

Flexibility applies to different components of a VM. To make the matter more manageable and to support the thesis without undue detour, this review will be restricted to issues pertaining to the object model and function-calling protocol.

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 [21]. The idea of characterizing a system in terms of four properties was also heavily inspired by their usage of the terms, although precise definitions were not provided.

The next section provides the historical roots of concepts belonging to each of the four properties. Finally, a comparison of our work to state-of-the-art systems related to JavaScript and instrumentation is made.

2.1 Historical roots

For each property of flexibility, the previous work will be presented in chronological order to provide a sense of evolution and temporal distance to the reader. As some of the work originates from the sixties, it should be clear to the reader that these ideas are not new. Throughout the exposition, we make explicit the elements that inspired our work and 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 will be required to obtain a desired behavior.

We will 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. The range of behavior modifications allowed by an implementation characterize the degree of openness.

The earliest account of the need for open implementations seems to be made by Shaw and Wulf in 1980 [26]. 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 [27], 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 3.2.1.2. 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 [17], 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 the fact that not only language implementations can be open but also applications [22]. Then, a methodology for designing open implementations was suggested by Maeda and al. [16], 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 [7]. 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++ [29].

Another major idea for extensibility was the notion of *encapsulation*, initially called *information* *hiding* by Parnas in 1972 [20]. 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' [9]. 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 [15] 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 [31] which later influenced the design of JavaScript.

The object representation presented in section 3.3 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 [18]. 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 [10] 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 [27], 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 [15] 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 will 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 [8]. 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 [6].

Both of these techniques were inspirations for the optimizations we explain in section 3.2.3.

2.2 Instrumenting JavaScript VMs

The last years have seen increasing interest from the research community toward JavaScript and web applications. Most relevant to our work here are empirical evaluations of JavaScript dynamic behavior. Ratanaworabhan and al. [23] as well as Richards and al. [25] have respectively shown that benchmarks do not correspond to the behavior of real Web applications and that JavaScript applications are more dynamic than what was previously thought. Both papers are sources of experiments to perform on JavaScript VMs and serve as a starting point in thinking about the capabilities required for a VM aiming for instrumentation. We replicate a part of the object model instrumentation from the second paper in our performance evaluation in section 5.3.

Richards and al. have also proposed another system, JSBench to record execution traces of JavaScript execution on websites, that can be replayed as standalone benchmarks [24]. Their system performs a source-to-source translation of JavaScript source code and proxies some operations of the language to record traces. Although they claimed the overhead introduced by their system for the benchmarks presented was less than 10% and depended on the number of proxy objects created, no comprehensive evaluation of the performance of their instrumentation strategy was done. It is therefore impossible to compare the performance overhead of our approach with theirs from the evaluation results they published. Further investigation would be required to see if our approach could be practical to record execution traces in the same manner.

Chapter 3

Design

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

Programming languages that survived the wilderness of actual usage and evolution usually suffer from birth or acquired quirks, defects and idiosyncrasies. JavaScript is no exception. In embarking on the journey of opening it for dynamic instrumentation, we faced the opposing goals of trying to be compatible with existing programs while keeping the design as simple as possible. This chapter explains our latest iteration, which runs existing benchmarks with no modification while providing the flexibility to change the semantics of object model operations and function calls *while the program is running.* The runtime environment required to implement the design can be written in 1719 lines of JavaScript code, satisfying our simplicity criteria.

The aim of this chapter is not to provide an exhaustive explanation of how we handled every idiosyncrasy of JavaScript. A reference implementation is provided for that purpose. Its *raison d'être* is to provide a high-level overview of the core ideas behind the system, hopefully in a way that will ease generalization to other languages.

Reification choices, namely the elements of the language we chose to open, are presented and their particular choice is justified. Then, the message-sending foundation is explained, both by seeing how the semantics of opened operation can be expressed on it and how it can be efficiently implemented in JavaScript. After, an object representation is presented. We believe this representation is original through its use of proxies that mirror the inheritance chain of the proxied objects. Finally, a brief compilation example is provided to illustrate how JavaScript can be compiled to this runtime environment.

[1] *Of course, all "there are only two" quotes have to be taken with a grain of salt.* [30]

3.1 Reification choices

Numerous elements of a programming language can be opened to user-modifications. A list relevant to JavaScript might include (but not be limited to):

- *Syntax*: How a program is represented (ex: as s-expressions, text or graphical tiles)
- *Control-flow operators*: The operators modifying the flow of control (ex: branching instructions, loops, exceptions, continuations)
- *Environment*: The list of variables accessed by a program and their scope (ex: local, global or closure variables)
- *Primitive operations*: Operations on primitive values for languages that have distinct primitive and object values (ex: `+` in JavaScript)
- *Object representation*: How objects are constructed from basic elements (ex: dictionary-based representation, fixed-memory blocks, etc.)
- *Object model operations*: What operations the objects support, what behavior they perform and how they affect their representation (ex: accessing a property, deleting a property, changing the class, etc.)
- *Function-calling protocol*: What operations are performed before and after a call. (ex: logging type information)

Our choice of which elements to reify has been influenced by past empirical studies and their choices of data to gather, by the availability of language features that support the reification and by the optimizations performed by the underlying runtime. Developments in any of those areas will further influence what elements are interesting, possible and practical to reify, therefore we urge the reader not to consider this work as the end of the road but instead as an example of what could be interesting, possible and practical given our current circumstances. The field is evolving, so should our understanding and practices.

Given our focus on instrumenting JavaScript, we left out the reification of the syntax. Our view is that a proper macro system would be ideal, but such a feature is not necessary to understand what current JavaScript programs are actually doing. Control-flow operators were left out, mostly for performance reasons but also because the current version of JavaScript would not allow their reification easily as a method. Wrapping a block in a function modifies its environment, potentially changing the **`this`** value in the block. The next version of JavaScript introduces an arrow function with a lexical **`this`** that will fix this short-coming, so as soon as modern VMs support the arrow-notation it should be possible to reify the control-flow instructions. The environment and primitive operations were kept closed for performance reasons, to ensure fast closure creation, variable access and arithmetic operations.

The object representation and the object model operations were reified because a large part of the dynamic instrumentation of JavaScript has targeted the object model to elucidate the amount of dynamism actually used. This reification was possible because the primitive operations are extensible through the `toString` and `valueOf` methods to support proxy objects. [2] The object representation and object model were designed to exploit the host's inline caches, global function inlining and calling protocol.

Finally, the function-calling protocol was reified to facilitate the construction of dynamic call graphs. It was possible because we can globally transform all functions for support. Our protocol was designed to avoid reliance on `call` [3] and `apply` native methods and to default to the native calling protocol as long as they are not modified, in the interest of speed.

3.2 Message-sending foundation

Simplicity is achieved when few abstractions can capture the essence of a whole class of phenomenons. In our design, the message-sending operation serves to unify all reified operations around a single mechanism. It facilitates the achievement of correctness, and arguably, performance, because it allows its implementation to target the highest performing operations of the underlying system. It would be unreasonable to expect better performance than a hand-tuned native implementation. [4] However, this approach might give us a better performance than equivalent instrumentations that try to optimize every reified operations independently, without the support of a unified mechanism.

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 eventually fast implementation. The initial awareness of the possibility to unify an object model around a message-sending primitive came from the Open, Extensible Object Models from Piumarta and Warth [21].

3.2.1 Semantics

We will now see the chosen semantics for the message-sending operation, in steps. We will first discuss the basic operation, that closely corresponds to a method call in JavaScript. We will then reify the function call to allow its redefinition. We will then see how we can reify the memoization operation performed

[2] It was a welcomed feature. It actually made our VM faster by permitting us to directly use the native primitive operators instead of also having to reify primitive operations to dispatch to our custom objects.

[3] When referring to the call method of a function, code typesetting will be used. When referring to function calls in general, normal typesetting will apply.

[4] However, faster-than-native speeds might be achieved if the native implementation is suboptimal and performs worse than a potentially equivalent other native functionality. It is my conjecture that as systems grow in complexity, such opportunities might be more common in the future.

by inline caches to enable user-supplied functions to specify their cached behavior, potentially allowing algorithmic improvements in speed. A message send is always synchronous to another object in the same address space, which corresponds to the most common usage in recent object-oriented languages.

To present the message-sending operation, we need to make some assumptions about the object representation. We will assume, without loss of generality, that the object representation is itself object-oriented, prototype-based and that operations to the representation are made by calling methods on objects. [5] However, the reader should keep in mind that the message-sending operation is orthogonal to the object representation chosen.

We will assume that the representation supports at least five 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 through the prototype chain
- `has(name)`: returns true if the `name` property is present on the object itself, without a lookup of the prototype chain
- `set(name, value)`: create a new `name` property if not already present, and set or update its value to `value`
- `getPrototype()`: returns the prototype of an object [6]

Unless specified, the pseudo-code follows the regular JavaScript semantics. We will take advantage of the expressivity of the rest and spread operators, bound to appear in the next revision [4]. For presentation simplicity, primitive values, missing values and error handling are omitted.

3.2.1.1 Basic

The basic version essentially performs a regular lookup followed by a method call. This is the semantics of a method call in JavaScript as well:

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

[5]In our particular case, it happens to be the case, as we will see later in this chapter.

[6]As explained in the conclusion, using the `__proto__` property breaks the encapsulation and mixes base-level and meta-level information. We therefore prefer to access the prototype through a method call. We use it here to advocate this particular usage.

From the point of view of the user program, the `call` operation is opaque and cannot be modified. When trying to obtain a dynamic call graph, this is unfortunate, because it forces the rewriting of every call site at compile time, including method calls, to catch all *call* events.

We can do it dynamically with a slight modification of the semantics of the operation.

3.2.1.2 Augmented function-calling flexibility

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 with proper support. We can also catch all functions calls, as long as these are mapped to message sends.

Performance *aficionados* might cringe at the idea of adding a level of indirection to every function call. Before we address performance issues of the message-sending protocol itself, we first look at a bigger opportunity of performance gains, namely memoizing the function behavior and see how we can support it.

3.2.1.3 Memoization protocol

Memoization is usually associated with functional programming and entails trading space-efficiency for time-efficiency by remembering past return values of functions with no side-effect. We will use the same definition here although we will relax the no-side-effect constraint by placing, on the programmer, the burden of insuring that memoized functions will always return the same value as if the original function was called. The programmer is expected to maintain system invariants when supplying memoization methods.

This particular functionality became necessary when trying to implement efficiently the JavaScript object model operations in Photon because they are reified as methods for openness. It became clear that using inline caches to memoize their optimized versions was desirable.

The basic idea is to allow 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 will be made to the memoized function.

There is an unfortunate interaction between memoization and the reification of the call protocol. A further refinement specifies that 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 will be ignored:

```
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(rcv, ..args);

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

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

}
```

This definition has the advantage that one can temporarily redefine the calling method without penalty after the original method has been restored.

Memoization introduced the idea of a cache for message sends. Caching entails invalidation if the invariants behind caching are violated. We will see how to handle that when discussing how to efficiently implement message sending. Before that, we will see how we can use message sending as a foundation for all reified operations.

3.2.2 Translating reified operations to message-sending

The following two sections summarize the operations of JavaScript that we opened by converting them to message sends, according to the semantics presented previously.

Function calls in JavaScript happen in various operations. An explanation of each occurrence as well as their equivalent message send are given in table 3.1.

An explanation of each opened object model operation as well as their equivalent message send are given in table 3.2.

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")`

3.2.3 Efficient implementation

Now that the semantics and mapping have been established, we will address the biggest hurdle that might prevent an approach based on a unified message sending foundation to be used, namely performance.

The core insight behind our implementation comes from seeing global function calls 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 is redefined at a later time, the call site will be 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 piggy back on those optimizations.

Let's start with an example. Given the aforementioned semantics of message sending, sending the message `msg` to an object `obj` inside a `foo` function can be written this way:

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

The `send` function is a global function. We can replace it with another global function that is guaranteed to be unique, so it can be identified with the call site. In addition to the message to be sent, we can pass it a unique identifier that will be used to find the corresponding global function name, for later specialization of the call site:

Table 3.2: Object model 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__")`

```
function initState(rcv, dataCache, ..args) {
    // Update ("codeCache" + dataCache[0])
    return send(rcv, dataCache[1], ..args);
}

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

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

Notice how the `initState` function follows the same calling convention as the `send` function. As we will see later, it is also the same calling conventions used for our function call protocol. Notice also that `dataCache0` is an array, which means that the different states of the cache can use the array to store additional information.

After an initial execution, the second time around, the cache will hold an optimized version of the operation. Given this new definition, the cache state might be equivalent to:

```
var codeCache0 = function (rcv, dataCache) {
    return rcv.get("msg").call(rcv);
};
var dataCache0 = [0, "msg"];

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

Apart from the indirection of the global function call, this example is optimal with regard to the object representation definition we have. If the underlying runtime chooses to inline the global function, the cost of the indirection will be effectively eliminated.

3.2.3.1 Cache states and transitions

We can build on the previous insight by precisely defining the behaviors of the inline caches and the conditions that trigger those behaviors. There are various ways to do it with different performance and flexibility profiles. We have not exhaustively explored the possibilities, therefore there might be interesting configurations yet to be uncovered. Nonetheless, we still found a simple arrangement that allows reified operations to be specialized to their call sites with a small memory overhead originating from tracking run-time invariants. An enthusiastic reader should read the following not as the final word on the matter but as a point of departure, as we believe there are other interesting configurations to be found.

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 will be invalidated. Also, if the `call` method on the `Function.prototype` object or any method with the `__memoize__` name is updated, *all* caches will be 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 will stay in the initial state and a regular message send will be 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.

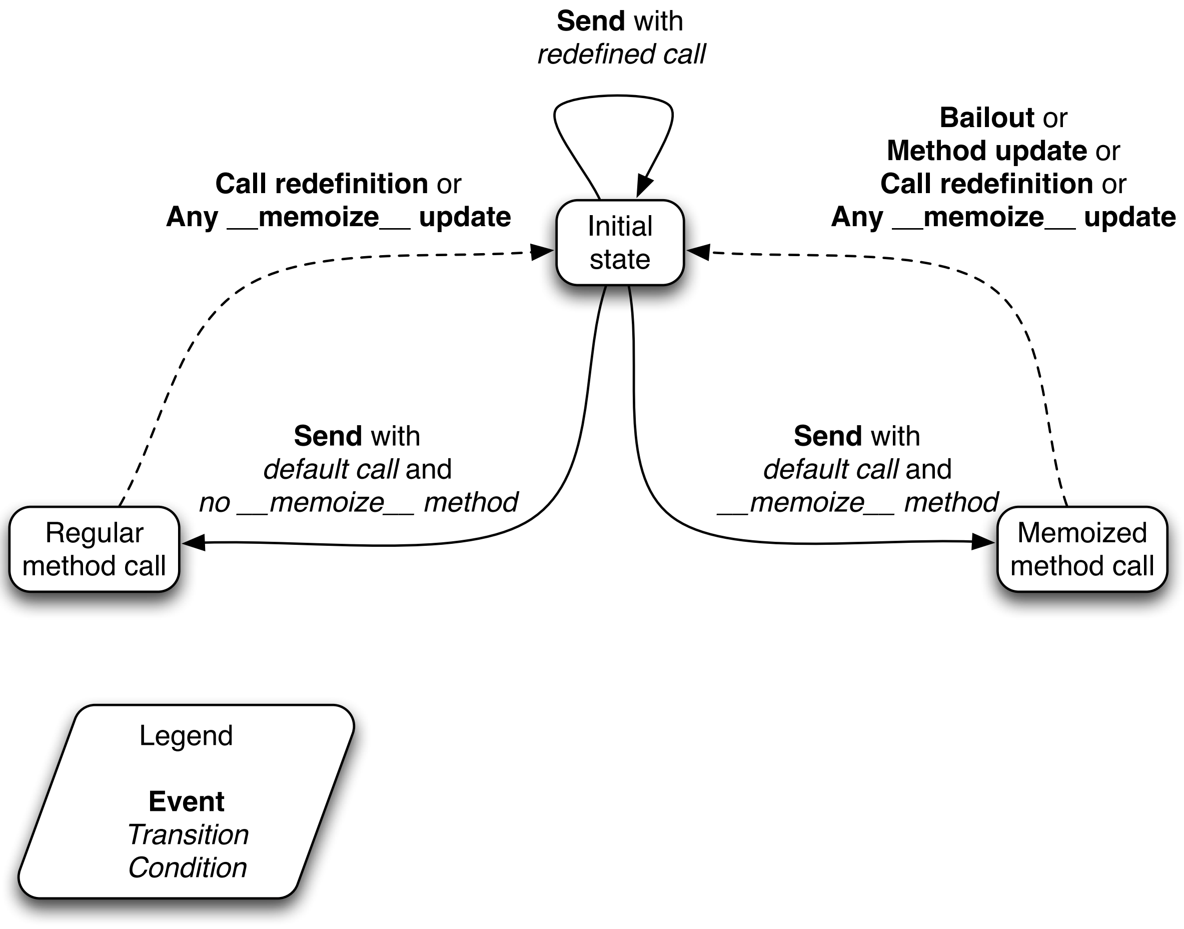
Send with redefined call
Bailout or Method update or Call redefinition or Any __memoize__ update
Call redefinition or Any __memoize__ update
Initial state
Send with default call and no __memoize__ method
Send with default call and __memoize__ method
Regular method call
Memoized method call
Legend
Event
Transition
Condition

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.

The initial inspiration for optimizing sends with global functions that would change during execution was drawn from the lazy function definition pattern as explained by Peter Michaux, in which after the initial setup performed by a function, a new function without the initialization code can replace the original function to provide an efficient operation [19]. We believe this is the first time this pattern is used as an inline cache in the literature.

For simplicity reasons, we did not try to aim for robustness in addition to flexibility and performance. It is therefore possible to break the system by supplying a memoized method that does not correctly maintain invariants. We decided to aim for maximum flexibility with a simple mechanism that could be used to specialize object model operations. The responsibility of supplying a correct method is left to the programmer. That being said, we believe it should be possible to further constrain this flexibility for robustness, within the same framework.

3.3 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 piggy backing of the underlying VM inline caches for performance. Our representation suggests promising optimization possibilities that would benefit from further exploration. We believe that our particular usage of the underlying object model dynamism is new and could serve as

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.

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, we should provide operations as method calls on objects whose internal representation is as close as possible to the host object being implemented. It can be achieved by structuring the object representation as proxies to native objects with a particular constraint, which we think has never been exploited before: *the proxy prototype chain should mirror the native objects' chain.*

The core idea is to associate a proxy object to every object in the system. In itself, this idea is not new. It has been advocated by Bracha and Ungar almost 10 years ago to provide meta-level facilities [5]. Proxy objects are also bound to appear in the next version of JavaScript and are currently supported by some major JS VMs. However to the best of our knowledge, it has never been suggested that the prototype chain of proxy objects should follow the prototype chain of proxied objects. As we will see, it opens the possibility of specializing and optimizing the object representation operations at runtime by attaching specialized methods at the appropriate places on the proxy prototype chain.

The basic object representation will be presented for native object types. We will then see how we can specialize methods for the number of arguments found at their call site. After, we will see how we can specialize operations for constant arguments values or specific object types.

3.3.1 Basic representation

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.

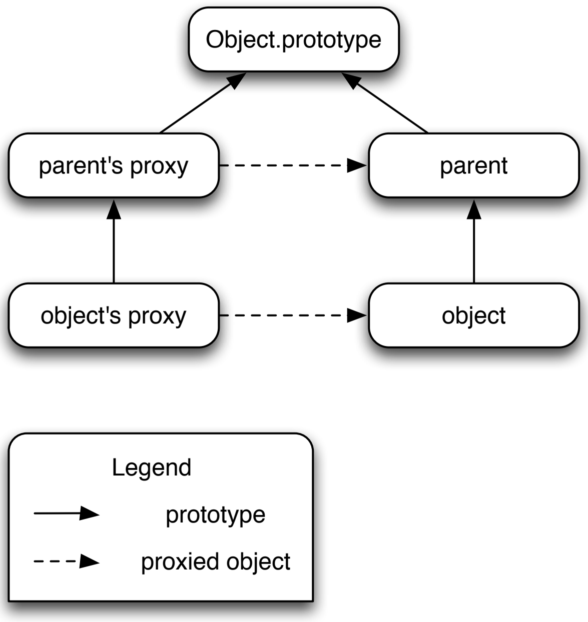

Figure 3.2: Basic object representation

An advantage of a representation like this one 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 basic 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. [7]

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

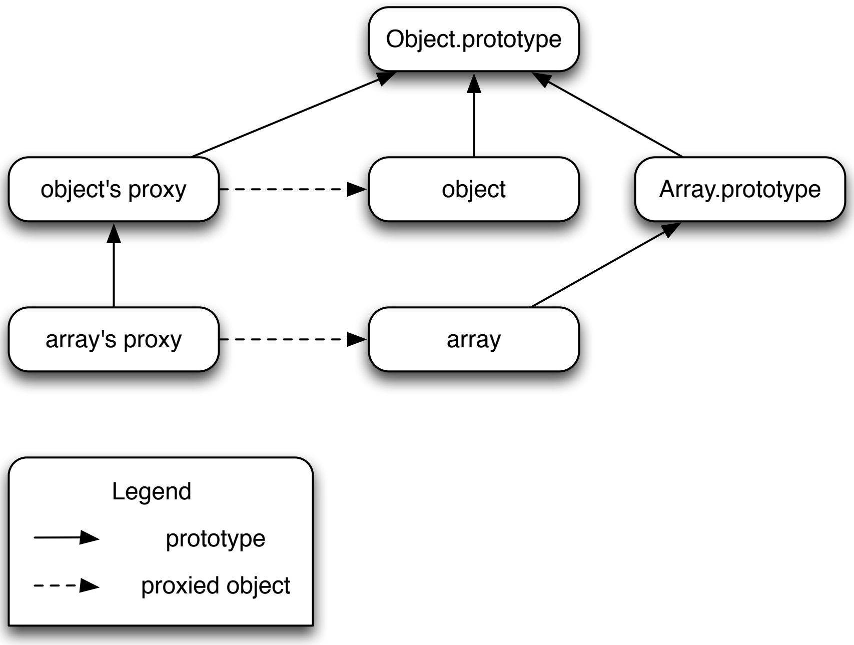

Figure 3.3: Special object representation

The astute reader will have noticed that although proxies mirror native objects in their inheritance chain, they do not mirror their properties. In fact, their properties can be fixed for the whole execution if one assumes that the semantics of object operations does not change. These facts are what allow proxies to adapt to dynamic circumstances by adding specialized methods at run time, which can be used for performance gains. We will see how to exploit this fact, first to specialize operations to a fixed number of arguments and then to a constant argument type or value. Those two specializations are independent and can be combined for further performance gains. 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.

3.3.2 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 could be:

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

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

// Also add a call1 method on all proxies
// that might be called as a method!
```

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 will then implement the specialized operation.

3.3.3 Type or value specialization

In the same way, we can specialize an operation for a known value. For example, suppose we would like an optimized operation that accesses a constant property name. A proper specialized method could be written:

```
var obj = root.create();

obj.set("foo", 42);

obj.get_foo = function () {
    return this.proxiedObject.foo;
};

root.get_foo = function () {
    return this.get("foo");
};
```

Notice that a method that falls back to the general case is provided to the root object to add support to all other proxies. The same idiom can be used for type specialization instead of value.

3.3.4 Note

The idea of mirroring the object prototype chain in the proxy prototype chain came as a surprise during the implementation of the VM and was partially exploited to obtain our current performance results. However, because of time constraints and not strict necessity to our thesis, an extensive empirical study of its performance characteristics and the exploitation of every optimization opportunity is yet to be performed. We decided to present the idea here first, before it was fully fleshed-out, to both claim originality and give the reader a primer on its possibilities. In the remainder of this dissertation, we will only present aggregate performance results for our VM on common benchmarks without trying to extract the contribution provided by this object representation. The reader should keep in mind that the arguments number specialization for the `call` method is performed but not the object-representation operation specialization. An exhaustive empirical study should be the focus of a subsequent publication.

3.4 Compilation example

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

- *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` will be 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 will show how we can use the added flexibility for instrumentation and runtime invariant monitoring.

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)`

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

Any system exhibits particular compromises between opposing goals. Some constraints derive from the nature of the problem a particular system is trying to solve. Different problems might have conflicting constraints. It follows that to assert the flexibility of a system, we need to choose particular use cases and leave out others. This chapter presents the uses cases we chose to support and how they can be implemented using our design.

We will see different examples modifying the base system to adapt it to particular use cases. The examples are written either in the implementation language of the VM, which has a direct access to the object representation and the execution environment or in the source language, which does not and correspond to the execution environment of user-supplied code. In other words, the implementation language is executed directly on the host VM and the source language is first translated to execute in a virtualized environment like regular source code. A comment line at the beginning of each example explicitly states if the example is written in the implementation language or the source language.

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 this 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 4.1 to model those relationships.

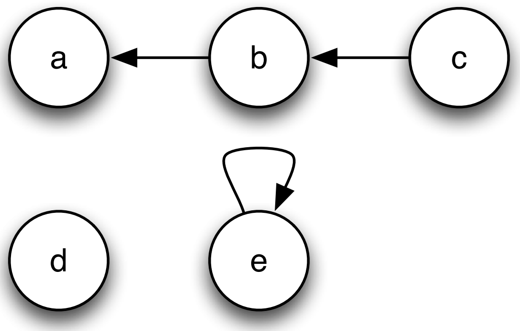

Figure 4.1: Call Graph for example code

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

[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.

function initializer redefines the `call` and `apply` methods on the root function and uses the `before` and `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 instrumentation. 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:

[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.

```
// 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);
```

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"));
})();
```

The next example performs the same operation but shows that this modification can be expressed in the source language instead:

```
// Language: Source JS
(function () {
    print("Retrieving original operation");
    var get = Object.prototype.__get__;

    print("Replacing the semantics of the operation");
    Object.prototype.__get__ = function (name) {
        var obj = this;

        while (obj !== null) {
            if (obj.hasOwnProperty(name))
                return get.call(obj, name);

            obj = obj.getPrototype();
        }

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

    print("Testing the semantics of the operation");
    try {
        ({}).bar;
        print("Should not be reached");
    } catch (e) {
        print(e);
    }

    print("Restoring the original behavior");
    Object.prototype.__get__ = get;
    print({}.bar);
})();
```

In both examples, 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, in the source 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 examples show the power of opening an implementation, beyond mere instrumentation. The fact they could both be written in the source language 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.

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 last section has shown how the additional flexibility provided by the system could be used to instrument the VM and modify the semantics of some of its operations. We now evaluate the actual run-time and memory-usage overhead incurred by the additional flexibility. The actual figures presented serve to evaluate the applicability of the approach to tasks other than instrumentation.

The next section explains the methodology used to evaluate the performance of the system. We then present the baseline performance and memory overhead obtained on different systems for common benchmarks. We then see the impact of a chosen instrumentation on run-time performance and memory usage and we conclude with an interpretation of the results.

5.1 Methodology

We compare three different systems:

- Photon running over V8 (V8's full optimizations)
- V8 (full optimizations)
- SpiderMonkey interpreter (JIT disabled)

V8 was chosen to host Photon because preliminary tests showed the system to be faster on it and V8 is known to be one of the fastest JS VMs. The additional speed is attributed to the ability of the runtime to do function inlining, on-stack-replacement and the presence of fast garbage collectors. Other VMs are catching up on features and the current focus seems to be on method-based Just-In-Time compilers so we anticipate that in the near future they could probably be used interchangeably. We compare Photon to V8 to quantify the performance cost incurred by our approach. We finally compare to a popular state-of-the-art interpreter, SpiderMonkey used in Firefox, to argue that our approach can be used wherever a manual instrumentation of that interpreter could have been performed.

In choosing the systems to compare, we eliminated a few current alternatives. We assume that when faced with the task of instrumenting production code to obtain run-time data, manually instrumenting a JIT-compiler would be deemed too complex to be cost-effective in terms of development time. At the time of writing, JavaScriptCore's low-level interpreter became available and replaced the original interpreter that was instrumented by Richards and al. [25] in WebKit. We argue that the only real instrumentation alternative right now would be SpiderMonkey's interpreter because the JavaScriptCore low-level interpreter is implemented in an assembly dialect to obtain performance gains. As this new interpreter matures, we speculate that its complexity will increase, negating most of the simplicity usually attributed to interpreters. [1] Finally, we do not show performance results for Narcissus, Mozilla's JavaScript in JavaScript interpreter, because the latest version would not run either SunSpider or V8 benchmarks.

To assess performance, we use the SunSpider and V8 benchmark suites, since they became the *de facto* standard to compare JS VM performance. They both come with their own different evaluation methodology. We used the original methodology of each benchmark suite to make our results comparable to other published results for other systems. All the benchmarks were run unmodified. However, the date benchmarks from SunSpider were omitted because they relied on using `eval` to access local variables, which is not supported by our current compiler. Since Photon's local environment uses the underlying VM's local environment (it is not reified), we believe that supporting this feature in the future will not significantly affect the performance results presented here.

We focus on two metrics, running time and the maximum heap size to respectively measure run-time performance and memory usage. Running time is measured either directly, in the case of SunSpider's benchmarks or indirectly through a score, in the case of V8's benchmarks. For V8's benchmarks, higher scores mean a faster system (less execution time). For SunSpider benchmarks, its the opposite, lower scores mean a faster system. Memory usage is used to estimate the overhead of Photon compared to V8, but it was not measured for the SunSpider interpreter because the information was hard to obtain.

[1]Should the reader still consider the alternative of instrumenting JavaScriptCore's low-level interpreter instead, she should know that in our tests, on V8's benchmarks, JavaScriptCore low-level interpreter was roughly three times faster than SpiderMonkey's interpreter. How much of those speed gains would remain in presence of instrumentation is unknown.

We present results in two different groups, the baseline performance and the instrumented performance. The baseline performance is used to measure the minimal overhead of the approach since it determines its viability, regardless of other characteristics. The instrumented performance is used to measure the impact of instrumentation on the baseline performance. It is common knowledge that instrumenting an interpreter has little impact over its overall performance (this was verified by Richards and al. when they instrumented JavaScriptCore [25]). However, our approach is more sensitive to instrumentation. We therefore quantify its impact.

The instrumentation chosen counts the number of occurrences at run time of each reified object-model operation, in the same spirit as the example given in section 4.1. The original operation is inlined in the replacement method instead of being called. The actual code is given in section 5.3 to show that the complexity is almost the same although we eliminated a function call. We chose this particular instrumentation because it is simple, it covers important object model operations and it was actually used to gather information about JS (it can be used to reproduce the object read, write or delete proportion figure from [25]).

All results were obtained on a MacBook Pro running OS X version 10.7.5 with a 2.2 GHz Intel Core i7 processor and 8 GB of 1333 MHz DDR3 Ram. We used V8 revision 12808 and SpiderMonkey version 1.8.5+ 2011-04-16. The results are intended to give an approximation of performance for the approach. Therefore, it was not deemed necessary to provide confidence intervals and averages of multiple runs, given that each test harness already did so. Although some variations between runs were noticed, they were sufficiently stable to not affect our argument.

For conciseness, abbreviations are used in the tables. They are listed in order of appearance:

- Pn: Photon
- SM: SpiderMonkey
- V8: V8
- Pn-spl: Photon with our simple instrumentation
- Pn-fast: Photon with an equivalent instrumentation optimized for speed

5.2 Baseline performance

5.2.1 Running times

The baseline performance for V8 benchmarks is shown in Table 5.1. It is pretty good compared to the SpiderMonkey interpreter for an overall score within a factor of 2. On three of the eight benchmarks

Photon is faster and in two cases by almost a factor of two. In other cases, the SpiderMonkey interpreter is between 2 and 3.5 times faster, except for the Splay benchmark where it is 13 times faster.

This last case seems to be a pathological case for the basic optimizations performed on property access, assignment and update. As will be seen in Table 5.5, for this particular benchmark, the instrumented version of Photon is five times faster than the non-instrumented version. This can be explained by the fact that the instrumented version *removes* the optimizations attempted. However, removing the same optimizations for all benchmarks gives an overall score 30% inferior with some benchmarks almost four times slower, except for Splay. Should we take the fastest score obtained without optimizations for Splay, Photon would be two times slower which is comparable to the other results.

Table 5.1: Baseline performance on V8 benchmarks

Benchmark	Pn	SM	V8	V8/Pn	SM/Pn
Crypto	1087.0	520.0	16829.0	15.5	0.5
DeltaBlue	145.0	409.0	18030.0	124.3	2.8
EarleyBoyer	647.0	1259.0	32965.0	51.0	1.9
NavierStokes	2632.0	811.0	19589.0	7.4	0.3
RayTrace	168.0	805.0	18333.0	109.1	4.8
RegExp	473.0	960.0	3803.0	8.0	2.0
Richards	142.0	330.0	14177.0	99.8	2.3
Splay	1286.0	1567.0	5904.0	4.6	1.2
V8 Score	500.0	737.0	13561.0	27.1	1.5

The baseline performance for SunSpider benchmarks, shown in Table 5.2 gives an overall score 1.6 times the SpiderMonkey interpreter. In this case, we use the geometric mean of the ratios because some pathological cases might give a wrong picture of the actual performance of the system if we were to simply use the total of all the running times to compute the overall ratios. 10 of 24 benchmarks are faster on Photon than on SpiderMonkey, with some as much as 5 times faster. This can be attributed to faster basic operations as compiled by the V8 JIT Compiler compared to the interpreted operations of SpiderMonkey. The slowest times are obtained on all string manipulation benchmarks such as crypto-* and string-*. One notable slow case is the string-tagcloud benchmark, which uses `eval` for JSON parsing, that calls into our compiler. The bottleneck seems to be the parsing stage of our OMeta-based compiler [32]. Experiments with other compilation techniques, still using JavaScript as an implementation language, were made in our lab with performance results than would make the compilation time negligible. For the rest, it seems that auto-boxing of strings and the wrapping of functions passed to the standard-library functions would be responsible for most of the slowdown.

5.2.2 Memory usage

The heap size is measured from garbage collection traces. When no garbage collection has been done, a result of '–' is given, which appears in some of V8 benchmark runs.

Memory usage for both benchmark suite in Table 5.3 and Table 5.4 look good given that every runtime object has an associated proxy with most cases using less than a factor of two in memory and two worst cases of 6.5 and 10. The high memory usage in EarlyBoyer might be explained by the memory overhead of every inline cache site having an associated array. In the case of string-unpack-code, the overhead might be explained by string and regexp proxies.

5.3 Instrumented performance

The instrumentation used for performance evaluation is a variation of the instrumentation given in section 4.1, that inlines the object model operation instead of wrapping it in another function. The simple version does not exploit the memoization protocol and corresponds to the straight-forward implementation:

```
var instrumentationResults;

(function () {
    var results = {
        __get__:0,
        __set__:0,
        __delete__:0
    };

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

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

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

    instrumentationResults = function () {
        var total = results.__get__ + results.__set__ + results.__delete__;
        return "Instrumentation results:\n" +
               "     __get__: " + results.__get__ + " " +
                    results.__get__ * 100 / total + "%\n" +
               "     __set__: " + results.__set__ + " " +
                    results.__set__ * 100 / total + "%\n" +
               "  __delete__: " + results.__delete__ + " " +
                    results.__delete__ * 100 / total + "%\n";
    }
})();
```

The fast version inlines the instrumentation operation inside the optimized version of object operations for speed:

```
var instrumentationResults;

var instrumentationData__get__ = 0;
var instrumentationData__set__ = 0;
var instrumentationData__delete__ = 0;

(function () {
    root.object.set("__get__", clos(function ($this, $closure, name) {
        instrumentationData__get__++;
        return $this.get(name);
    }, (function () {
        var getLength = clos(new Function("$this", "dataCache", "name",
            "instrumentationData__get__++\n"+
            "return $this.getLength();"
        ));

        var get = clos(new Function ("$this", "dataCache", "name",
            "instrumentationData__get__++\n"+
            "return $this.get(name);"
        ));

        var getNum = clos(new Function ("$this", "dataCache", "name",
            "instrumentationData__get__++\n"+
            "return $this.getNum(dataCache, name);"
        ));


        return clos(function ($this, $closure, rcv, method, args, dataCache) {
            var name = args.get(0);
            if (rcv instanceof ArrayProxy && (typeof name) === "number") {
                if (options.verbose) print("Caching __get__ numerical at " + dataCache.get(0));
                return getNum;
            } else if (name === "length" && dataCache.get(2)[1] === "string") {
                if (options.verbose) print("Caching __get__ length at " + dataCache.get(0));
                return getLength;
            } else {
                if (options.verbose) print("Caching __get__ at " + dataCache.get(0));
                return get;
            }
        });
    })()));
```

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

    var ownedNames = {};
    function updateProperty(name) {
        if (!hasProp(ownedNames, name)) {
            ownedNames[name] = clos(new Function ("$this", "dataCache", "name", "value",
                "if ($this.map === dataCache[3]) {\n" +
                "   instrumentationData__set__++\n"+
                "   if (tracker.hasCacheLink(name)) tracker.flushCaches(name);\n" +
                "   return $this.payload."+name+" = value;\n" +
                "}\n" +
                "return bailout($this, dataCache, name, value);"
            ));
        }
        return ownedNames[name];
    }

    var newNames = {};
    function createProperty(name) {
        if (!hasProp(newNames, name)) {
            newNames[name] = clos(new Function ("$this", "dataCache", "name", "value",
                "if ($this.map === dataCache[3]) {\n" +
                "   instrumentationData__set__++\n"+
                "   if (tracker.hasCacheLink(name)) tracker.flushCaches(name);\n" +
                "   $this.map = $this.map.siblings[name];\n" +
                "   return $this.payload."+name+" = value;\n" +
                "} return bailout($this, dataCache, name, value);"
            ));
        }
        return newNames[name];
    }

    var set = clos(new Function ("$this", "dataCache", "name", "value",
        "instrumentationData__set__++\n"+
        "return $this.set(name, value);"
    ));

    return clos(function ($this, $closure, rcv, method, args, dataCache) {
        var name = args.get(0)
        var cacheId = dataCache.get(0);

        if (dataCache.get(2)[1] === "string" &&
            name !== "__proto__" &&
            rcv.set === root.object.set) {
            if (rcv.map.properties[name] === true) {
                return updateProperty(name);
            } else {
                return createProperty(name);
            }
        } else {
            return set;
        }
    });
})()));
```

```
    root.object.set("__delete__", clos(function ($this, $closure, name) {
        instrumentationData__delete__++;
        return $this.delete(name);
    }));

    instrumentationResults = function () {
        var total = instrumentationData__get__ +
                    instrumentationData__set__ +
                    instrumentationData__delete__;
        return "Instrumentation results:\n" +
               "     __get__: " + instrumentationData__get__ + " " +
                    instrumentationData__get__ * 100 / total + "%\n" +
               "     __set__: " + instrumentationData__set__ + " " +
                    instrumentationData__set__ * 100 / total + "%\n" +
               "  __delete__: " + instrumentationData__delete__ + " " +
                    instrumentationData__delete__ * 100 / total + "%\n";
    }
})();
```

The first version is intended to measure what performance can be expected from a quickly developed instrumentation while the second one 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.

5.3.1 Running times

Table 5.5 and Table 5.6 show the impact of instrumentation on the baseline performance. In both cases, the fast version has negligible impact with some results slightly faster, mostly due to natural variation of results. However, the simple version can be as much as 30 times slower on some benchmarks.

5.3.2 Memory usage

Table 5.7 and Table 5.8 show that on V8 and SunSpider benchmarks, the simple and fast instrumentations have no impact on memory usage, with the exception of the Splay benchmark in the simple case. This might be caused by our tracking strategy that has an entry for every cached regular method call. This might be removed not tracking certain method names such as `__get__`, `__set__` and `__delete__` and conservatively invalidating all caches when they are redefined.

5.4 Interpretation

A conclusion can be made that our current optimizations are not sufficiently stable across benchmark results and a better predictor of whether to optimize or not would give more even results, especially for V8's Splay benchmark. We believe this can be done within the framework presented. Further, the memory overhead seems acceptable for instrumentation tasks given our multi-gigabytes of RAM in today's machines. Instrumentation can have a negligible impact both on performance and memory usage, as long as the operations instrumented are optimized enough. Otherwise, the choice of trading performance for a faster development time can still be made with an impact on performance that can be reasonable in most cases.

The reader should keep in mind that our focus has been first and foremost on bringing flexibility to JS and as explained in the design chapter, some known optimizations can still be applied for further possible gains that are yet to be quantified. We therefore argue that the performance obtainable by our approach is *at least* what we present here but the upper bound is still to be found and will be the object of further research. The conclusion presents promising ideas for improving both the baseline performance and instrumented performance.

Table 5.2: Baseline performance on SunSpider benchmarks

Benchmark	Pn	SM	V8	Pn/V8	Pn/SM
3d-cube	6.7	1.3	0.3	22.3	5.2
3d-morph	0.4	0.1	0.1	4.0	4.0
3d-raytrace	7.4	1.3	0.5	14.8	5.7
access-binary-trees	0.6	0.1	0.1	6.0	6.0
access-fannkuch	0.5	0.1	0.1	5.0	5.0
access-nbody	2.8	0.4	0.2	14.0	7.0
access-nsieve	0.6	0.1	0.1	6.0	6.0
bitops-3bit-bits-in-byte	0.4	0.1	0.1	4.0	4.0
bitops-bits-in-byte	0.4	0.1	0.0	Infinity	4.0
bitops-bitwise-and	0.0	0.0	0.0	NaN	NaN
bitops-nsieve-bits	0.5	0.1	0.1	5.0	5.0
controlflow-recursive	1.0	0.1	0.1	10.0	10.0
crypto-aes	4.2	1.0	0.3	14.0	4.2
crypto-md5	4.5	0.7	0.3	15.0	6.4
crypto-sha1	3.1	0.4	0.2	15.5	7.8
date-format-tofte	2.2	0.5	0.2	11.0	4.4
date-format-xparb	5.6	0.6	0.4	14.0	9.3
math-cordic	1.3	0.1	0.1	13.0	13.0
math-partial-sums	0.7	0.1	0.1	7.0	7.0
math-spectral-norm	1.0	0.1	0.1	10.0	10.0
regexp-dna	0.0	0.0	0.0	NaN	NaN
string-base64	0.8	0.1	0.1	8.0	8.0
string-fasta	1.0	0.1	0.1	10.0	10.0
string-tagcloud	3.5	0.4	0.3	11.7	8.8
string-unpack-code	0.0	0.0	0.0	NaN	NaN
string-validate-input	1.0	0.1	0.1	10.0	10.0
SunSpider Score	50.3	7.9	3.8	13.2	6.4
Geometric Mean	0.0	0.0	0.0	NaN	NaN

Table 5.3: Baseline memory usage on V8 benchmarks

Benchmark	Pn	V8	Pn/V8
EarleyBoyer	123.0	20.0	6.2
NavierStokes	28.0	19.0	1.5
crypto	55.0	20.0	2.8
deltablue	32.0	20.0	1.6
raytrace	35.0	20.0	1.8
regexp	60.0	22.0	2.7
richards	28.0	18.0	1.6
splay	155.0	97.0	1.6

Table 5.4: Baseline memory usage on SunSpider benchmarks

Benchmark	Pn	V8	Pn/V8
3d-cube	27.0	18.0	1.5
3d-morph	21.0	18.0	1.2
3d-raytrace	28.0	17.0	1.6
access-binary-trees	21.0	17.0	1.2
access-fannkuch	22.0	–	–
access-nbody	23.0	18.0	1.3
access-nsieve	21.0	–	–
bitops-3bit-bits-in-byte	21.0	–	–
bitops-bits-in-byte	21.0	–	–
bitops-bitwise-and	21.0	–	–
bitops-nsieve-bits	22.0	17.0	1.3
controlflow-recursive	21.0	–	–
crypto-aes	28.0	17.0	1.6
crypto-md5	26.0	–	–
crypto-sha1	27.0	17.0	1.6
math-cordic	22.0	–	–
math-partial-sums	22.0	18.0	1.2
math-spectral-norm	22.0	–	–
regexp-dna	31.0	–	–
string-base64	29.0	17.0	1.7
string-fasta	27.0	17.0	1.6
string-tagcloud	–	19.0	–
string-unpack-code	175.0	19.0	9.2
string-validate-input	26.0	18.0	1.4

Table 5.5: Instrumented performance on V8 benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn/Pn-spl	Pn/Pn-fast
Crypto	1087.0	50.8	1019.0	21.4	1.1
DeltaBlue	145.0	54.6	144.0	2.7	1.0
EarleyBoyer	647.0	236.0	635.0	2.7	1.0
NavierStokes	2632.0	58.4	2427.0	45.1	1.1
RayTrace	168.0	97.0	174.0	1.7	1.0
RegExp	473.0	346.0	475.0	1.4	1.0
Richards	142.0	37.6	133.0	3.8	1.1
Splay	1286.0	448.0	1237.0	2.9	1.0
V8 Score	500.0	110.0	485.0	4.5	1.0

Table 5.6: Instrumented performance on SunSpider benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn-spl/Pn	Pn-fast/Pn
3d-cube	6.7	7.0	7.0	1.0	1.0
3d-morph	0.4	0.4	0.4	1.0	1.0
3d-raytrace	7.4	7.6	7.5	1.0	1.0
access-binary-trees	0.6	0.7	0.7	1.2	1.2
access-fannkuch	0.5	0.6	0.5	1.2	1.0
access-nbody	2.8	3.0	2.9	1.1	1.0
access-nsieve	0.6	0.6	0.6	1.0	1.0
bitops-3bit-bits-in-byte	0.4	0.4	0.4	1.0	1.0
bitops-bits-in-byte	0.4	0.4	0.4	1.0	1.0
bitops-bitwise-and	0.0	0.0	0.0	NaN	NaN
bitops-nsieve-bits	0.5	0.6	0.5	1.2	1.0
controlflow-recursive	1.0	1.2	1.0	1.2	1.0
crypto-aes	4.2	4.3	4.3	1.0	1.0
crypto-md5	4.5	4.6	4.5	1.0	1.0
crypto-sha1	3.1	3.1	3.1	1.0	1.0
date-format-tofte	2.2	2.1	2.2	1.0	1.0
date-format-xparb	5.6	5.6	6.1	1.0	1.1
math-cordic	1.3	1.5	1.2	1.2	0.9
math-partial-sums	0.7	0.7	0.7	1.0	1.0
math-spectral-norm	1.0	1.1	1.1	1.1	1.1
regexp-dna	0.0	0.0	0.0	NaN	NaN
string-base64	0.8	0.8	0.8	1.0	1.0
string-fasta	1.0	1.0	1.0	1.0	1.0
string-tagcloud	3.5	3.5	3.5	1.0	1.0
string-unpack-code	0.0	0.0	0.0	NaN	NaN
string-validate-input	1.0	1.1	1.1	1.1	1.1
SunSpider Score	50.3	52.0	51.8	1.0	1.0
Geometric Mean	0.0	0.0	0.0	NaN	NaN

Table 5.7: Instrumented memory usage on V8 benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn-spl/Pn	Pn-fast/Pn
EarleyBoyer	123.0	123.0	123.0	1.0	1.0
NavierStokes	28.0	33.0	29.0	1.2	1.0
crypto	55.0	58.0	55.0	1.1	1.0
deltablue	32.0	32.0	32.0	1.0	1.0
raytrace	35.0	34.0	34.0	1.0	1.0
regexp	60.0	56.0	59.0	0.9	1.0
richards	28.0	28.0	27.0	1.0	1.0
splay	155.0	150.0	152.0	1.0	1.0

Table 5.8: Instrumented memory usage on SunSpider benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn-spl/Pn	Pn-fast/Pn
3d-cube	27.0	27.0	27.0	1.0	1.0
3d-morph	21.0	21.0	21.0	1.0	1.0
3d-raytrace	28.0	27.0	27.0	1.0	1.0
access-binary-trees	21.0	21.0	21.0	1.0	1.0
access-fannkuch	22.0	22.0	22.0	1.0	1.0
access-nbody	23.0	23.0	23.0	1.0	1.0
access-nsieve	21.0	21.0	21.0	1.0	1.0
bitops-3bit-bits-in-byte	21.0	21.0	21.0	1.0	1.0
bitops-bits-in-byte	21.0	21.0	21.0	1.0	1.0
bitops-bitwise-and	21.0	22.0	22.0	1.0	1.0
bitops-nsieve-bits	22.0	22.0	22.0	1.0	1.0
controlflow-recursive	21.0	21.0	21.0	1.0	1.0
crypto-aes	28.0	28.0	28.0	1.0	1.0
crypto-md5	26.0	31.0	26.0	1.2	1.0
crypto-sha1	27.0	27.0	27.0	1.0	1.0
math-cordic	22.0	22.0	22.0	1.0	1.0
math-partial-sums	22.0	22.0	22.0	1.0	1.0
math-spectral-norm	22.0	22.0	22.0	1.0	1.0
regexp-dna	31.0	31.0	31.0	1.0	1.0
string-base64	29.0	30.0	32.0	1.0	1.1
string-fasta	27.0	25.0	27.0	0.9	1.0
string-tagcloud	–	100.0	100.0	–	–
string-unpack-code	175.0	179.0	177.0	1.0	1.0
string-validate-input	26.0	26.0	26.0	1.0	1.0

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 at a baseline performance within a factor of two of a state-of-the-art interpreter, by harnessing a sophisticated run-time optimizer, 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 on functions the environment information. 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 [25], 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 [32] 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 baseline performance of the system and the performance while the code is instrumented. We briefly touch upon each of them.

6.2.3.1 Improve the baseline performance

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.

The other technique to be tried is the object representation method specialization presented in the design section. Further gains could be had if both the lookup and the argument specialization were combined in a single method.

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] Google Traceur. `http://code.google.com/p/traceur-compiler/`, December 2012.

[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] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 2004 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] D. Ingalls. Design Principles Behind Smalltalk. *Byte*, 6(8), August 1981. Special Issue on Smalltalk `http://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html` – geprüft: 3. May 2012.

[12] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.

[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] 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.

[16] 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.

[17] 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.

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

[19] P. Michaux. Lazy function definition pattern.
`http://michaux.ca/articles/lazy-function-definition-pattern`, December 2012.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] F. Story and S. Vajirā. *Last days of the Buddha: the Mahā Parinibbāna Sutta*. Buddhist Publications Society, 1988.

[29] 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.

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

[31] 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.

[32] 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.