Reification of Execution State in JavaScript

Implementing the Lively Debugger

Christopher Schuster

A thesis submitted in partial fulfillment
of the requirements for the Degree of
Master of Science

Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Germany

Supervisors:
Prof. Dr. Robert Hirschfeld
Jens Lincke

April 2012

Contents

List of Figures

List of Listings

List of Tables

Acronyms

API Application Programming Interface. xv, 13, 17, 18, 21, 25, 43, 57, 58, 64, 68

AST Abstract Syntax Tree. iv, vii, 13, 22, 29–31, 33, 34, 36–38, 40, 42, 46

COP Context-Oriented Programming. 25, 45, 51, 52

CSS Cascading Style Sheets. 13

DOM Document Object Model: Object-oriented interface for interacting with the browser. 13, 16, 57

HTML HyperText Markup Language [4, 5]. 1, 13, 14, 16

IDE Integrated Development Environment: Software system with editor, debugger, project and build management. 13

IR Intermediate Representation: Machine-generated source code representation which is suitable for optimization and further processing. 12, 13, 22, 36, 37

JIT Just-in-time compilation occurs during the program runtime as opposed to ahead-of-time (AOT) compilation done by traditional compilers.. 7, 8, 12, 53

PC Program Counter: A special register storing the memory address of the next instruction to execute. iv, ix, 3, 5, 26, 36, 37, 39, 40, 48, 62, 64

SP Stack Pointer: A special register storing the memory address of the current top element on the stack. 3

SSA Static Single Assignment. ix, 12, 26, 40, 61–63

SVG Scalable Vector Graphics [39]. 16

TTD Test-Driven Development [57]. 46

UML Unified Modeling Language (UML) [3]. vii, 30

VM Virtual Machine: A virtual environment abstracting the underlying hardware and platform. iii, vii, xv, 6, 7, 11, 13, 21–24, 27, 31, 33, 38, 40, 52, 53, 59, 62, 64, 67

Abstract

As web pages evolve to complex software applications, the web authoring tools must evolve as well. The Lively system integrates development and deployment of applications into a collaborative, self-sustaining environment on the web. However, the JavaScript debugging tools in Lively are still incomplete due to the lack of a cross-browser debugging API that allows access to the runtime execution state without relying on plugins.

We achieved execution state reification by interpreting JavaScript on top of the underlying JavaScript VM. This does not affect the runtime performance of Lively because the interpreter is only used for code that explicitly requires access to the execution state. Seamless transitions between natively executed and interpreted code enable on-demand debugging with breakpoints and stepping.

The implementation of this approach showed that closures and other JavaScript features like dynamic evaluation cause problems when both modes of execution work together. These problems were solved for Lively by restricting these language features but a general purpose cross-browser debugger is topic of future work.

Zusammenfassung

Die Fortschritte bei den Webtechnologien haben mittlerweile dazu geführt, dass aus einfachen Webseiten komplexe Softwareanwendungen wurden. Die Werkzeuge zum Entwickeln dieser Anwendungen müssen sich somit auch weiterentwickeln. Das Lively-Projekt hat sich zum Ziel gesetzt, Webinhalte und -anwendungen auch direkt kollaborativ im Web zu erstellen aber wird hierbei maßgeblich von den fehlenden Browser-Schnittstellen bei der Bereitstellung dieser Entwicklungswerkzeuge behindert. Gerade die Fehlersuche in den Webanwendungen kann momentan nur durch Browser-Plugins realisiert werden.

Wir haben diese fehlenden Schnittstellen nun bereitgestellt, indem wir einen JavaScript Interpreter entwickelt haben, der innerhalb des Browsers Code ausführen und insbesondere auch anhalten kann und es somit ermöglicht, den aktuellen Ausführungszustand zur Fehlersuche zu nutzen. Dieser Interpreter führt allerdings nicht dazu, dass die Ausführungsgeschwindigkeit von Lively leidet, da er nur bei Bedarf genutzt wird, also beispielsweise zur schrittweisen Ausführung des Programms. Die Ausführung wechselt hierbei zwischen dem Interpreter und JavaScript VM des Browsers.

Bestimmte Konzepte der JavaScript-Programmiersprache, wie Funktionen mit gekapseltem Zustand und die dynamische Codeausführung, haben sich dann bei der Implementierung als problematisch erwiesen für die Zusammenarbeit dieser beiden Ausführungsmodi und werden nicht unterstützt. Die im Rahmen dieser Arbeit entwickelten Werkzeuge zur Fehlersuche funktionieren somit nicht für beliebigen JavaScript-Code. Sie können aber für Lively effektiv eingesetzt werden.

Acknowledgements

I owe my deepest gratitude to Robert Krahn for his ongoing support and especially for his work on stack reification in Lively on which this thesis is based. I am also very thankful to my supervisor, Jens Lincke, and my professor, Robert Hirschfeld, for many insightful discussions and their guidance with the research and the writing of this thesis. Additionally, I would like to thank Tobias Pape who helped me with his expertise in typography.

Lastly, this thesis would not have been possible without the support and understanding of my friends and family.

Christopher Schuster

Chapter 1

Introduction

Over the last two decades, web pages evolved from simple documents with texts, images and forms to complex interactive applications. The web browser still uses HyperText Markup Language [4, 5] (HTML) for presentation, but it also supports interaction and dynamic behavior by means of embedded JavaScript[1] code.

The wide availability of JavaScript across all kinds of devices makes it the language of choice for programming web applications which have the advantage of not needing local installation, being platform-independent, being immediately globally available over the Internet, while at the same time protecting the user from malicious code by running in a sandbox with limited capabilities. For these reasons, "the web is displacing proprietary operating systems as the next application platform"[7].

The Lively project goes one step further in this direction by providing an integrated environment that is both source and target platform for applications and content on the web. Being written in itself, the self-sustaining Lively environment is programmed incrementally and relies on continuous testing and debugging to eliminate programming errors. This makes sophisticated development tools like debuggers essential. In fact, the general lack of adequate debugging tools across all programming languages is also known as the "debugging scandal" [8].

All debugging tools have in common that they support the developer in finding and understanding the cause of failures or unintended behavior by narrowing down the number of possible locations of this *bug* in the source code. The Lively debugging tools are unaware of the developer's intentions and therefore focus on letting developer observe the execution of a program and inspect its *execution state* at different points in time.

[1]JavaScript is a dialect of ECMAScript [6] used in all major web browsers.

This execution state, however, is managed internally by the JavaScript engine and not accessible in a uniform way across different browsers.

> "It is lamentable that the JavaScript standard provides almost no access to the runtime execution state [...] and the ability to resume a suspending computation."
>
> Ingalls et al. [9]

The implicit execution state of the running program has to be made explicit, it has to be *reified*, so that JavaScript tools that can access and manipulate it without depending on plugins or other browser-specific interfaces. The reification of the execution state in order to implement debugging tools for Lively is the topic of this thesis.

Outline

The chapters 2, 3 and 4 are explaining basic concepts and definitions necessary to understand the following chapters.

- Chapter 2 describes what the term *execution state* refers to in different environments and how it can be accessed.
- Chapter 3 deals with the JavaScript language in particular and to what degree modern web browsers enable execution state reification.
- Chapter 4 describes the Lively environment and its existing development tools.

The main part of this thesis focuses on the way JavaScript execution is altered in Lively to access the execution state and to provide a debugger capable of halting, resuming and stepping.

- Chapter 5 describes the approach of using a JavaScript interpreter to reify the execution state while restricting it to relevant code to limit the performance overhead.
- Chapter 6 outlines the design and implementation of this approach including the integration into the Lively environment.

The last part of the thesis sets the work into context and concludes the thesis.

- Chapter 7 discusses the results and experiences with execution state reification as described in the previous chapters.
- Chapter 8 points to related work.
- Chapter 9 considers possible future extensions and research ideas based on this work.
- Chapter 10 summarizes the lessons learned.

Chapter 2

Execution state

At each point in time a program currently in execution has a certain state. This **execution state** is partly explicitly defined in the programming language, e.g. with local variables, and partly held by the hardware, the operating system, and the runtime environment.

Programs are usually written to serve a certain purpose or to solve a specific problem in the domain the user is interested in. The internals of the underlying execution are not part of this domain and therefore only implicitly assumed in the program source code. Some programs, however, are concerned about their own execution or the execution of other programs for reasons like logging, tracing, profiling, security or debugging. In these cases, the execution state itself is part of the domain. To enable this type of *meta-programming*, the execution state needs to be explicitly addressable in the program, the execution state must be **reified**.

There are different ways of execution state reification depending on the level of abstraction.

2.1 Hardware level

Computers designed in accordance with the *Von Neumann architecture* [10] execute software by treating parts of the data in memory as instructions. These instructions operate on registers, memory and hardware devices. Therefore the execution state in the context of a processor consists of the registers, the memory, which is divided in the **heap** and the **stack**, the **program counter (PC)**, **stack pointer (SP)** as well as other special registers depending on the architecture.

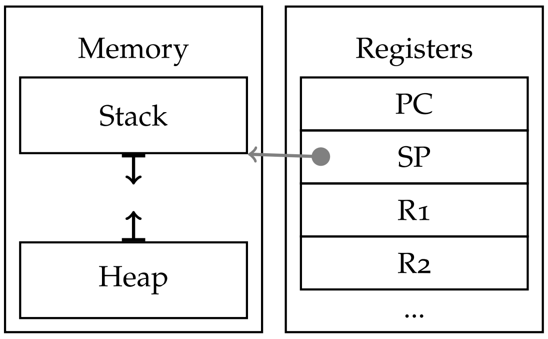


Figure 2.1: Execution state in hardware

Execution State reification can either be done by accessing the maintenance interfaces of the hardware with external devices or by using special processor instructions for debugging if the processor offers them.

2.2 Operating system level

Software running on top of a **multi-tasking** operating system is restricted in its interaction with the underlying hardware and other programs. This is necessary to allow the operating system to manage shared resources of different simultaneously running programs. All of these programs have their own execution states. Such an execution state is not simply identical to the execution state of the processor. Instead, it is managed by the operating system and also includes the internal state of the operating system with regard to the program, such as open file handles, quota, etc.

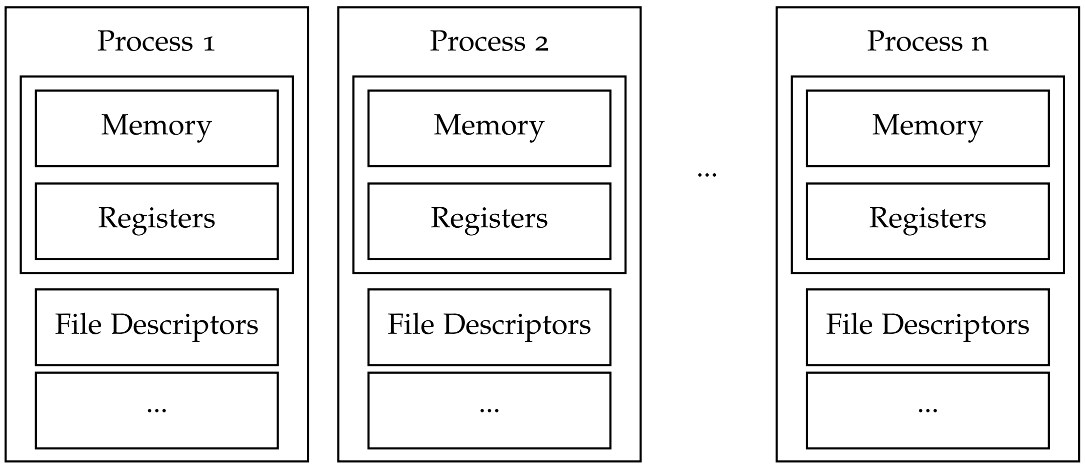


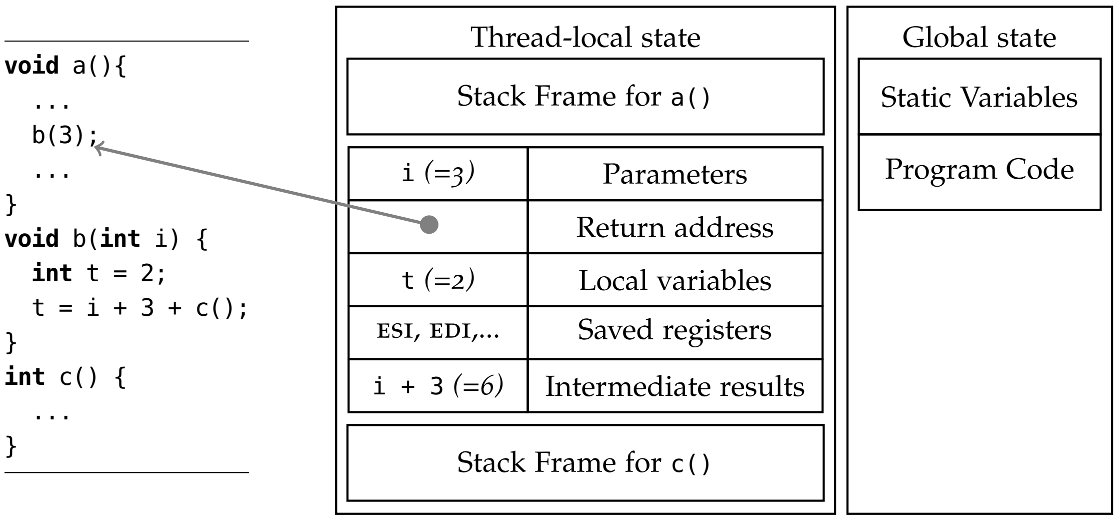
Figure 2.2: Execution state in a multi-tasking operating system

Therefore programs that deal with the execution state of other programs, such as debuggers, need to be aware of the operating system and the state held by it for the program in question.

2.3 High-level languages

High-level programming languages offer abstractions to help the programmer. They advocate variables instead of registers, structural programming instead of branching and function calls instead of manual stack manipulation. This means that the execution state needs to be reified on the same level of abstraction.

Function calls branch to a subroutine with the ability to return to the call-site. This is done by saving and restoring the state of the current thread of execution on the stack. Each of these **stack frames** includes the return address, as well as arguments, local variables, temporary variables and register values. Shared state that is non-local to the thread is considered global and not stored on the stack.



```
void a(){
  ...
  b(3);
  ...
}
void b(int i) {
  int t = 2;
  t = i + 3 + c();
}
int c() {
  ...
}
```

Figure 2.3: Execution state in the C language according to the calling convention of the Intel x86 processor architecture [1]. The processor is currently computing the new result for `t` and has not returned from `c()` yet.

2.3.1 Compiled Code

If the language is compiled to machine code, high-level abstractions are usually lost and the generated code is unreadable for the user. This is why debuggers for compiled languages like the GDB[1] require the compiler, e.g. GCC[2], to emit additional *metadata* as shown in Listing 2.2[3]. Among this information are symbolic names for variables and functions (hence the name *symbolic debugger*) as well as the mapping between instruction data and line numbers in the source code. Combined with the native execution state of the program it is now possible to represent the execution state in terms of the high-level programming language, e.g. the PC can be a line in the source code instead of a memory address.

2.3.2 Interpretation

Another way to execute a program written in a high-level language is to interpret the code at runtime. The interpreter is using the program's source code as input

[1]The GNU Debugger is an open-source source-level debugger for C, C++ and other languages [11]
[2]GNU Compiler Collection [12]
[3]There multiple standards for debugging metadata, e.g. DWARF by the Free Standards Group [13]

```c
void b(int i) {
  int t = 2;
  t = i + 3 + c();
}
```

Listing 2.1: Example C program

```
    .globl  b             // function name
    .type   b, @function
b:
    .loc 1 1 0            // line 1:
    ...                   //   callee entry
    .loc 1 2 0            // line 2:
    movl    2, -8(%ebp)  //   t = 2
    .loc 1 3 0            // line 3:
    movl    8(%ebp), %eax //   s0 = i
    leal    3(%eax), %ebx //   s1 = x + 3
    call    c             //   s2 = c()
    addl    %ebx, %eax    //   s2 = s2 + s1
    movl    %eax, -8(%ebp) //   t = s2
    .loc 1 4 0            // line 4:
    ...                   //   callee exit
    ret                   //   return
```

Listing 2.2: Debug information in generated Assembler code for Listing 2.1.

and performs actions and calculations according to the semantics of the interpreted language. With this approach decisions can be deferred to the runtime of the program, making it more dynamic and flexible.

Because the interpreter controls every part of the program's execution, the execution state of the interpreted program can be completely independent from the underlying hardware and operating system, e.g. an interpreter might use a **Spaghetti Stack** data structure for the call stack which, in contrast to the hardware stack, simplifies resuming after exceptions etc.

On the one hand this means that the execution state can easily be accessed and manipulated (as shown in Section 5.1.3), but on the other hand this also means that the execution state can only be reified inside the program by explicit interaction with the interpreter.

2.3.3 Virtual machines

Some languages have abstractions and concepts that require a virtual machine (VM) for execution. A VM uses either compilation, interpretation or a combination of both to execute code and additionally offers memory management, scheduling, foreign-function interface, code management, etc.

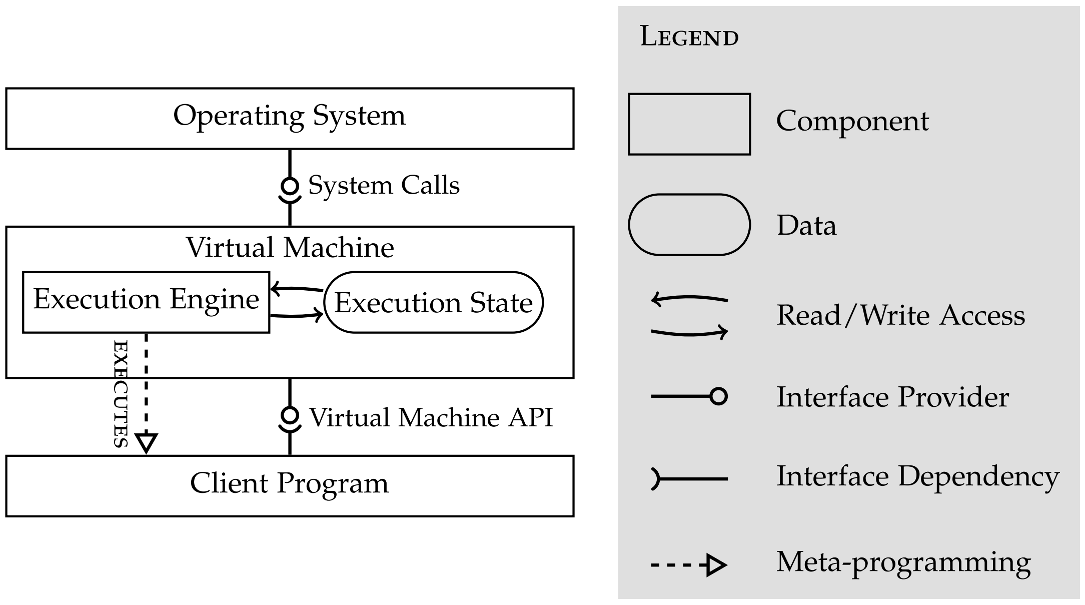


Figure 2.4: Basic virtual machine components

Execution Sate inside the VM

Depending on the language and the concrete implementation these additionally features may affect the execution state of the running program, e.g. a program that called a native function of the environment may be suspended while the VM is waiting for the completion of the call.

Supporting multiple execution engines

More importantly, the combination of **interpretation** and **compilation** leads to challenges for execution state reification. VMs like the Java HotSpot [14] VM improve the runtime performance of the program by just-in-time (JIT) compilation of repeatedly executed (*hot*) loops to highly optimized machine code. This means that parts of the execution state are tied to the hardware and operating system, similar to compiled languages (see Section 2.3.1), while other parts of the execution state are mainly managed by the interpreter.

Sharing execution state between engines

The transition between different execution engines requires a conversion of parameters and return values from one stack layout to another one which causes a performance overhead. Additionally, users of the programming language should not be aware of this optimization technique and would be surprised if the execution state is not reified during the native execution of generated machine code.

One solution would be to use exactly the **same stack layout** for both the interpreted as well as the compiled execution. This, however, limits possible optimizations during compilation. Ideally, the JIT compiler is able to use type information gathered during runtime profiling or static analyzes, e.g. *type inference*, to generate code that omits certain type checks and uses processor instructions operating on native values. The interpreter, in contrast, makes no assumptions on the types and depends on *type-tagged values* for every operation.

Another solution, proposed by the developers of Firefox JägerMonkey [15], was to use **two stacks**. One of these two stacks would include everything shared by both the compiled and the interpreted code, while type tags and other metadata can be held on the other stack for use by the interpreter. Section 5.3 deals with this problem in the context of JavaScript execution in a browser and describes the solution implemented by the Lively debugger.

Chapter 3

Execution state in JavaScript

3.1 JavaScript language

JavaScript is an imperative, object-oriented, prototype-based [16], dynamically-typed programming language with C-like syntax. The first JavaScript implementation was included in the *Netscape Navigator 2.0* in 1995 [7]. At first, JavaScript was primarily used for simple scripts on web pages, e.g. validating input fields in forms, but since then more complex applications were written in JavaScript even outside the web browser, e.g. server-side code with *node.js* [17] or mobile applications with *Callback* [18].

A full description of the programming language is not subject of this thesis. The purpose of this chapter is rather to point out key concepts and distinct features of the language that are relevant to the following chapters.

3.1.1 Expressions and control flow

JavaScript has expressions with operator precedence which are similar to the expressions of the C language and Java. But in contrast to many other languages, expressions with questionable semantics do not raise an exception, cause an error state, or result in undefined behavior. The JavaScript language is **very permissive**, **error-tolerant** and uses special return values instead.

The examples in Table 3.1 illustrate how the weak type system of JavaScript automatically convert operands, e.g. an empty array is converted to an empty string and adding two empty strings is done by concatenation. However, there are also some examples in which types cannot be used interchangeably and generate errors, e.g. calling a function by adding braces to an identifier, e.g.`myFun()` will fail if the type of `myFun` is not a function. Also, referencing an undefined global variable will generate an error rather than return `undefined` except if the expression is an assignment, e.g.`myGlob = 1`.

Expression	Code	Return Value
Division by Zero	`23 / 0`	`Infinity`
Zero divided by Zero	`0 / 0`	`NaN`
Modulo Zero	`23 % 0`	`NaN`
Accessing a non-existent property	`anObject.p`	`undefined`
Adding two undefined values	`undefined + undefined`	`NaN`
Adding two null values	`null + null`	`0`
Adding two empty arrays	`[] + []`	`""`

Table 3.1: Faulty JavaScript expressions and their return values

There are also expressions with explicit control flow. For example the ternary operation will first evaluate the condition expression and then depending on the result either evaluate and return the second or the third expression, e.g. `return a>2?a:2`.

3.1.2 Function calls

Functions are called using the same syntax as in C or Java. The major difference is, that there are actually four different ways to call a function.

1. If the expression belonging to the function call is accessing the property of an object, e.g.`obj.method()`, then the property will be treated as method of the object and the keyword `this` will be bound to the object.
2. If the `new` keyword was used right ahead of the call, e.g.`new Date()`, then the function is assumed to be a constructor and will be called with a new object bound to `this` which will have the same *prototype* as the function to enable prototypical inheritance.
3. The value of `this` can also be explicitly set by using the indirect function calls `apply` or `call`.
4. In all other cases, the function will be called with `this` bound to the global object, e.g. the browser window.

Another noteworthy characteristic of JavaScript in a browser environment is the **event-driven** execution model. Conceptually there is always only one thread of execution which runs inside the main event loop of the web page. This means that the web page cannot be rendered and input cannot be handled during the execution of long-running function calls. To prevent this from happening, the amount of computation has to be limited and time-consuming functionality like input/output needs to be done asynchronously. This also means that simply halting the execution within a function call makes the web page completely unresponsive. Chapter 5 explores ways around this limitation and Section 7.2.3 discusses the potential consequences of breaking the event-driven model.

3.1.3 Scoping

The previous section already explained the special dynamic scoping rules of `this` in JavaScript. With that exception all other variable names have lexical scopes. This means that the *static* position of the variable name in the source code with regard to its definitions defines the variable binding.

The binding is done on function-level instead of block-level, so all uses of a variable name within a function naturally refer to the same variable. This basically has the same effect as *hoisting* all definitions to the top of the function as in Listing 3.1.

```
function f() {
  return x;   // ReferenceError: x is not defined
}

function g() {
  return x;   // No problem,
  var x;      // because x is defined down here
}

function h() {
  var x = 0;
  for (var x in {a:2}) {
    // ...
  }
  return x;   // Returns the string "a"
}
```

Listing 3.1: Scoping in JavaScript

The lexical scoping allows accessing the variables of the surrounding function even if the function has already returned. The state of the variables is enclosed in the function and thus makes it a **closure**. Listing 3.2 shows a closure whose state is not visible by the rest of the code.

3.2 Major JavaScript implementations

In contrast to other VMs, JavaScript implementations used in browsers have very hard latency requirements because the web page rendering is paused every time a JavaScript snippet is encountered and needs to be parsed and evaluated.

```
function counter() {
  var i = 0;
  return function() { return ++i }
}
var count = counter(); // returns the closure
count();   // increments i
count();   // increments i again
           // but there is no other way to access i anymore
```

Listing 3.2: Closures in JavaScript

The currently most popular open-source JavaScript implementations are *JägerMonkey*, which is currently used by the Firefox browser, *JavaScriptCore* [19], which is part of WebKit and used by the Safari browser, and *V8* [20], the JavaScript engine of the Chrome browser.

3.2.1 JägerMonkey

JägerMonkey [15] is the current JavaScript execution engine of the Firefox browser. It uses a combination of interpretation and JIT compilation. Only repeated execution of a function triggers compilation because the compilation process initially consumes memory and time and improves performance only in the long run. Additionally, certain JavaScript language features like `eval` are very hard to compile to native code.

Up until November 2011, Firefox also included a tracing JIT called *TraceMonkey* which recorded traces through the code and generated native code for fast paths based on the recorded traces. This proved to be less efficient than using a whole method JIT compiler, which can also be applied to code with many trace and type combinations [21], and led to JägerMonkey as the successor of TraceMonkey.

JägerMonkey uses a technique called *inline threaded code* which is similar to *threaded code* [22] but inlines hand-coded assembly templates for each instruction of the intermediate representation (IR). These templates are not generated at runtime which results in a shorter compilation time but also prevents optimizations across templates. Additionally, only the common case gets inlined and must be protected by a guard condition, typically a type check. To achieve this, all values are type-tagged by using a technique called *NaN boxing* [7, 23] and the type needs to be checked for each instruction. Future generations of the Firefox JavaScript execution engine, like the proposed **IonMonkey**, specifically target this problem by using static single assignment (SSA) as IR and applying standard compiler optimizations like *code motion*, *common subexpression elimination* and *register allocation* [7].

3.2.2 V8

The V8 JavaScript engine [20], which is used by Google Chrome and the open-source Chromium, was developed under the technical lead of Lars Bak who also contributed to other VMs, e.g. the Self and Strongtalk implementation, and the HotSpot Java VM. This explains certain design decisions made in V8 and similarities between these VMs like the *polymorphic inline caching* of V8 [24] which was first introduced in Self [25] and *tagged integers* [23, 26] which are common in Smalltalk implementations [27].

In contrast to JägerMonkey and other JavaScript implementations, V8 does not use an interpreter for code execution. Instead, every function in the JavaScript source code is directly compiled to native code for the target platform. Sophisticated compiler optimizations based on *data-flow analysis* and *type inference* are done on the level of the abstract syntax tree (AST); no IR like *bytecode* is used.

JavaScript application run by V8 use the machine stack just like other compiled programs (see Section 2.3.1). Due to the compilation, many abstractions of JavaScript are lost on this low level of execution which means that metadata must be maintained by the VM to reify the execution state. Furthermore, V8's dynamic optimization techniques like *inline caching* generate optimized code at runtime that does not correspond to any JavaScript function in the source code.

3.3 Debugging

There are many tools available to support JavaScript application developers. Because JavaScript is primarily used within browsers, JavaScript development tools are often interacting with a browser environment. This integration can either be achieved by embedding a browser into the integrated development environment (IDE), e.g. *Eclipse* [28], or by enhancing the browser with additional *development plugins*. Firebug [29] and the Chrome developer tools [30] are examples of the latter and very popular among JavaScript developers.

3.3.1 Firebug

Firebug is a plugin for the Firefox browser that provides Document Object Model (DOM) inspection, dynamic HTML and Cascading Style Sheets (CSS) manipulation as well as halting the JavaScript execution and debugging the scripts. There is also a cross-browser version called Firebug Lite which uses only standard JavaScript application programming interfaces (APIs) but has a limited feature set.

> "Other features though are too dependent in browser internals and will not be supported (at least in a near future), such as the Javascript debugger."
>
> Firebug Lite Developers [31]

Chapter 5 describes a solution to implement a cross-browser debugger without relying on browser internals.

3.3.2 Logging

Without the help of debugging tools, JavaScript developers typically resort to logging, also known as `printf` *debugging* due to the infamous C language function, to display the runtime execution state. This requires inserting logging statements into the source code and reading the output by using a JavaScript console, e.g. when using the `console.log` interface, a logfile, message boxes with the `alert` function, or even writing HTML to the current web page [32].

Chapter 4

Lively development tools

The Lively environment, also known as *Lively Kernel*, was originally developed at Sun Inc. and has been documented by Ingalls et al. [9] among others. The source code and a live public instance, called *Webwerkstatt*, is available at `http://lively-kernel.org/`. The design and implementation of Lively are not subject of this thesis. Instead, this chapter summarizes basic concepts of Lively that were applied to build a cross-browser debugger on the basis of execution state reification.

4.1 The Lively environment

The dynamic nature of JavaScript, which is covered in Chapter 3, enables the user to adapt and change a system at runtime. This means that the application can simultaneously be modified and executed which is not possible with compiled languages.

Such a development style is called *exploratory programming* and requires that the development tools are part of the running system. This ultimately leads to an integrated *self-supporting* environment [9] where there is no clear boundary between development tools and running application [33].

Lively uses the ubiquitous JavaScript language which is available on every platform and operating system. Therefore, in contrast to similar environments like Smalltalk [34] or Self [35], Lively does not need to be installed locally, it does not rely on plugins, and it has collaboration built in from the start with easy distribution, deployment and upgrade of applications.

4.2 Class system

Despite the fact that JavaScript has a prototype-based inheritance, as mentioned in Section 3.1.2, the Lively codebase is organized in classes and methods similar to the class system in Smalltalk. Lively enables polymorphic inheritance by providing a pseudo variable named `$super` which automatically gets passed as an argument to each method and enables invoking the implementation of the parent class.

Additionally, Lively uses a module system to group classes, extensions, etc. together into units which are loaded on demand and which declare dependencies. Unfortunately, browser vendors have not agreed on a standard for a module system yet, so Lively uses a custom implementation similar to the proposal for *CommonJS Modules* [36].

```
module('games.Dice').requires().toRun(function() {
  Object.subclass('games.dice.Game',
  'initializing', {
    initialize: function() { this.result = 0; }
  },
  'accessing', {
    getResult: function() { return this.result; }
  },
  'gambling', {
    rollDice: function(n) { this.result += Math.floor(Math.random() * 6 + 1); }
  });
});
```

Listing 4.1: `Dice.js`: A module for a dice rolling game

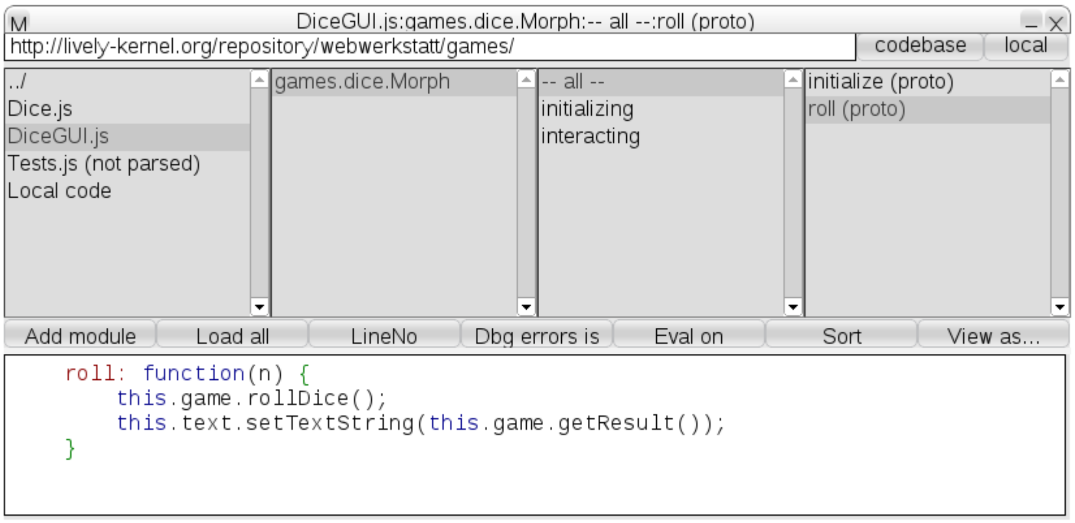
A minimal Lively module consisting of one class with two methods, an instance variable and a constructor is shown in Listing 4.1. Such a module is usually saved in a file, e.g.`Dice.js`, and edited by using a *system code browser* as shown in Figure 4.1. The source of a method, class or module can be browsed, edited, evaluated, and saved back to a remote code repository. This enables seamless, collaborative web authoring without external tools.

The system code browser supports syntax highlighting but there is still room for improvement when it comes to static analysis, code generation, *refactoring*, autocompletion, and static error detection like accessing uninitialized variables or unreachable code.

4.3 Morphic

For graphics and user interface generation, Lively uses a reimplementation of *Morphic* which was first implemented in Self [37] and later ported to Squeak [38].

The actual rendering is abstracted from the developer, e.g. Lively used Scalable Vector Graphics [39] (SVG) in the past, switched to the DOM and there are even plans to support *HTML5 Canvas*.



```
    roll: function(n) {
        this.game.rollDice();
        this.text.setTextString(this.game.getResult());
    }
```

Figure 4.1: System code browser displaying the module in Listing 4.1

4.3.1 Writing Morphic applications

In Morphic, the whole graphical environment is composed of **morphs** which are objects with graphical appearance (shape, color), geometrical properties (position, size), behavior (event handling, stepping, etc.), and submorphs (to create a scene graph). Morphs can be manipulated visually as real objects, i.e. the user can pick up, move, drop, resize and clone morphs at runtime [40].

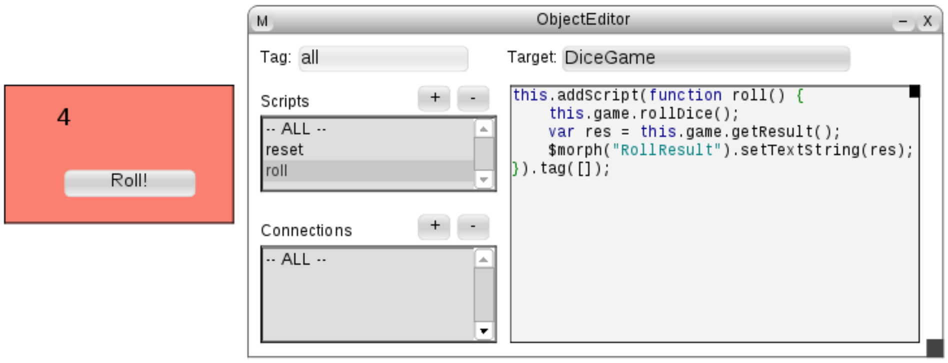
System code browser

One way to write Morphic applications is to use the system code browser and create classes following the same style API as in Listing 4.1. This is usually done by subclassing Morphic classes, modifying its behavior with the *template method pattern* [41] and using instance variables to store references to other morphs.

Object editor

Alternatively, Morphic applications can also be created without writing classes at all by composing objects instead.

Simple geometrical shapes, widgets and even complex components, which are stored in a collaborated *PartsBin*, can be used as templates and basic building blocks. By composing and visually manipulating these objects an instance of the application can be adjusted to fit the needs. Assigning names to morphs avoids the need for referencing instance variables, and visual *connections* replace callbacks and explicit event handlers.



```
this.addScript(function roll() {
    this.game.rollDice();
    var res = this.game.getResult();
    $morph("RollResult").setTextString(res);
}).tag([]);
```

Figure 4.2: Writing Morphic scripts with the object editor

Instead of using a system code browser dynamic behavior is added to morphs with an *object editor*. As shown in Figure 4.2, a morph can have *scripts* attached which are comparable to class methods but tied to a concrete morph instance. A morph with all its properties, scripts and submorphs can then be stored in the PartsBin for use by other users and developers.

4.4 Debugging

While the system code browser and the object editor target writing code, the developer also needs tools to run and debug the code.

4.4.1 Logging

The simplest way to display the execution state is to add logging statements manually into the code which has been discussed in Section 3.3.2. Lively offers a logging API to display messages and errors both in the Lively user interface as well as on a separate console window.

It is also possible to trace the execution without altering the code by using *method wrappers* which can also measure the execution time and are discussed in Section 5.3.1.

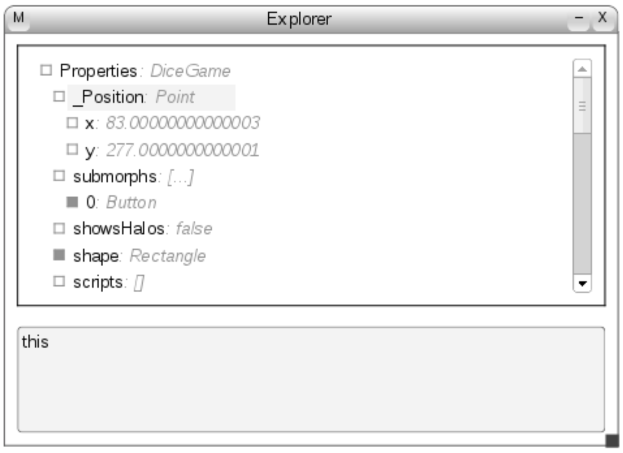
4.4.2 Displaying execution state

Lively provides a hierarchical object inspector and a call stack viewer which can be used to inspect the state of objects and call stacks. These tools, however, are only available as long as the system is not currently executing a function due to the restrictions given in Section 3.1.2.

4.4.3 Observing execution

Ideally, Lively would also be able to halt the program at any given point in the source code, inspect the execution state with all the tools described in the previous section, and resume the execution step-by-step like other systems with fully integrated tool support, e.g. Squeak [38].

Prior to this thesis, this was only possible by using browser-specific tools like Firebug (see Section 3.3.1). These tools do not benefit from the other Lively tools and are not aware of the Lively module system which results in inaccurate source code locations, method names etc.[1] Fortunately, cross-browser execution state reification makes it possible to implement these missing debugging tools (see Section 6.7 and 6.7.5).



```
☐ Properties: DiceGame
  ☐ _Position: Point
    ☐ x: 83.00000000000003
    ☐ y: 277.0000000000001
  ☐ submorphs: [...]
    ■ 0: Button
  ☐ showsHalos: false
  ■ shape: Rectangle
  ☐ scripts: []
```

```
this
```

Figure 4.3: Hierarchical object inspector

[1]There are efforts to get a better integration of dynamically evaluated code and custom languages into browsers by the user of *source maps* [42]

Chapter 5

Partial interpretation

In order to provide sophisticated development tools for Lively, including a debugger, the execution state of a running JavaScript program needs to be reified. The best way to achieve this is to access the execution state at the level of the JavaScript VM. However, as of today there is no unified cross-browser API to access the execution state of the underlying JavaScript VM directly because implementations like JägerMonkey and V8 follow different approaches of executing JavaScript (see Section 3.2) and therefore have different execution state representations.

This chapter outlines different ways to reify the execution state and enable debugging in Lively despite these differences by not depending on a concrete JavaScript implementation.

5.1 Interpreting JavaScript

Without making any assumptions about the underlying implementation of the language, it is still possible to have an explicit execution state if the code in question is executed by a client VM on top of the host VM. Such a client VM targets the same language it accepts as input which influences the implementation of the its execution engine and memory management.

5.1.1 Execution engine

When choosing between the two strategies for executing code, compilation and interpretation, interpretation is preferable because it is easier to implement and enables accessing the execution state on a higher level of abstraction (see Section 2.3.2).

Both the client VM and the host VM execute JavaScript but from now on the client VM will be referred to as **interpreter** and the host VM will simply be called **JavaScript VM**.

A simple interpreter would parse JavaScript source, convert it into a semantically equivalent IR, e.g. an AST or bytecode, which can be either register- or stack-oriented, and then execute one *AST node* or *bytecode instruction* at a time by performing the appropriate actions.

Except for the interpreter itself all the code must be interpreted in order to have an explicit execution state that does not rely on the JavaScript VM.

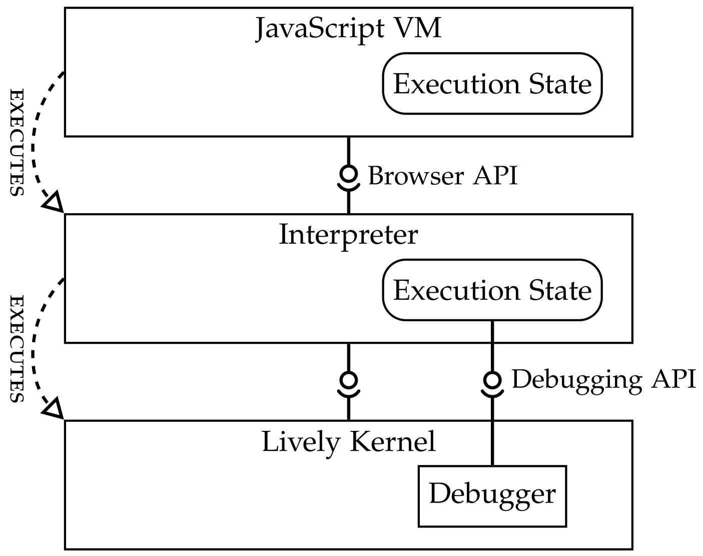


Figure 5.1: Using an interpreter on top of the JavaScript VM to enable execution state reification. *This diagram uses the same notation as Figure 2.4.*

5.1.2 Memory management

In order to support high-level language features a VM provides abstractions for state and data that are implemented on top of the target platform and operating system. There is usually a substantial amount of complexity involved in this mapping, but a VM that runs on top of the same language it executes is able to delegate almost all of the abstractions directly to the underlying platform.

Since the interpreter itself is written in JavaScript and executes JavaScript code, it can reuse the memory management, the data structures, and most of the built-in functionality like type conversion and primitive operations from the browser's JavaScript VM. This significantly simplifies the implementation of the interpreter.

5.1.3 Accessing the execution state

Except for shared objects, like the global namespace, the execution state of the interpreted JavaScript code is completely separated from the actual JavaScript VM implementation. Because the interpreter controls every aspect of the execution state, it can also provide an interface for the interpreted program to access its own execution state (see Figure 5.1) and thereby achieve execution state reification.

5.2 Limiting the scope of interpretation

The approach given in the previous section works well but also introduces a considerable performance penalty for the whole system. The execution state is necessary for debugging but irrelevant for other use cases which will run significantly slower and not benefit from the interpreter.

Following the general practice of optimizing for the common case, it would be better to run code natively on the JavaScript VM as long as the execution state is not accessed. This requires the code executed by the interpreter to seamlessly integrate with the code executed by the JavaScript VM and vice versa. The execution state of the interpreted code will be available for debugging purposes while the rest of the code runs fast on the JavaScript VM. This partial interpretation is depicted in Figure 5.2.

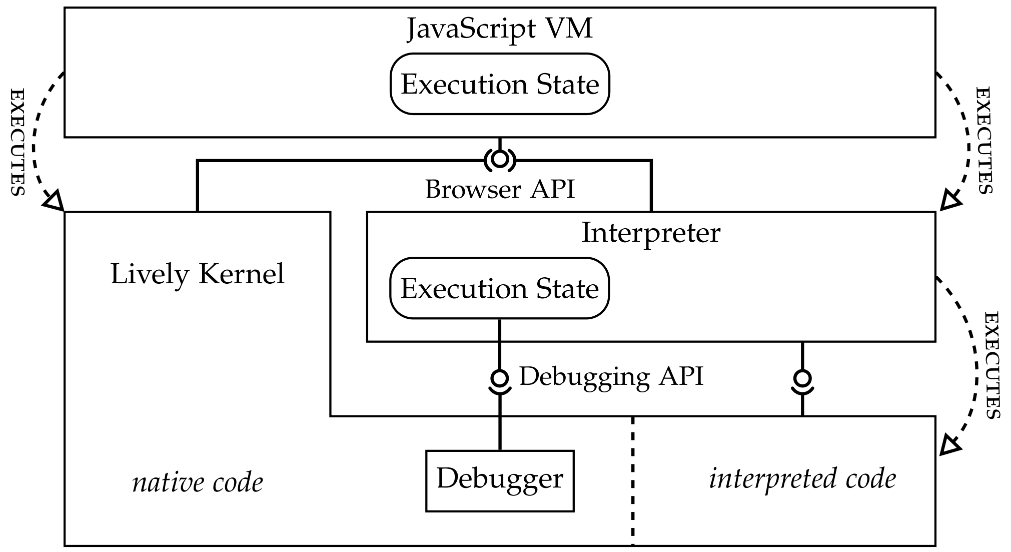


Figure 5.2: By interpreting parts of the Lively environment the debugger is able to access the execution state of the interpreted code at runtime. *This diagram uses the same notation as Figure 2.4.*

5.2.1 Identifying relevant execution state

The difficulty now lies in deciding which portions of the code need execution state reification. There is no way to generally anticipate the program's and user's intent but it is often sufficient for debugging purposes to let the user decide when to enable interpretation. For example, with native execution being the default execution mode, the following rules could be used to trigger a transition into the interpretation mode.

- Functions containing source code breakpoints, i.e.`debugger` statements, will always be interpreted regardless whether the breakpoint will actually be reached or not.
- When code is explicitly evaluated by the user, it will be executed either natively or by the interpreter depending on the keyboard command used to trigger the evaluation.
- Certain user-initiated events, like mouse button clicks while holding a special debug button, will cause the interpreter to execute the associated event handler instead of the JavaScript VM.
- Morphs, objects, classes or whole namespaces can be marked as debugging subjects whose scripts and methods are always interpreted.
- Code that was just written by the user and has never been executed before is more prone to bugs than other parts of the system. By using the interpreter for its very first execution errors can be reported with more details about the execution state than otherwise.

5.2.2 Debugging scope generally undecidable

The last rule in the previous section refers to one of the most common debugging use cases: halting the execution when a critical error or unhandled exception occurs. Ideally, the user would be able to inspect the detailed execution state of the faulty statement, the surrounding function, and all the stack frames above. But this is only possible if the interpreter was used for the function as well as all the functions calling it. According to Rice [43], there is no general way to decide whether a function will result in error prior to evaluating it, and therefore the on-demand interpreter cannot debug all general errors. This fundamental problem always occurs whenever a user or a debugging tool needs to make a distinction between faulty and correct code:

> "The user can't make the decision about whether to see the details of an expression if he or she doesn't know whether this expression contributes to the bug. This leaves the user in the same dilemma as the instrumentation tools – they must have a reasonable hypothesis about where the bug might be *before* they can effectively use the debugging tools!"
>
> Lieberman and Fry [44]

5.3 Execution state outside interpreter

If any of the rules listed in Section 5.2.1 applies and the execution switches to interpretation mode, the execution state available to the user for debugging purposes will be limited to the stack frames executed by the interpreter. It might be possible to display function names and approximate line numbers by using non-standard browser APIs but information about local variables and intermediate computation results will be lost.

Due to the event-driven model of JavaScript, the natively executed stack frames had already been terminated by the time the user interacts with development tools on the web page. This means that the user can step through the interpreted portion of the code and resume its execution but natively executed stack frames are generally not resumable.

To complement the interpreter and support execution state reification for natively executed code, techniques like *method wrappers* and *source code transformations* can be used.

5.3.1 Method wrappers

Method wrappers follow a relatively simple principle and are common in languages like Smalltalk [45, 46]. They change the reference to the original method in the method lookup table of the target class to a wrapper that executes code before and after the invocation while having access to the arguments and the return value.

Therefore they are ideal for implementing *aspect-oriented programming*, *context-oriented programming (COP)*, *tracing* and *profiling*. By not modifying the JavaScript source, they can even be used for built-in functions not implemented in JavaScript. Compared to interpretation, the performance overhead caused by method wrappers is small enough that it is feasible to instrument every part of the Lively environment.

The main drawback of method wrappers is the lack of information about local variables and intermediate computation results. In the context of debugging, method wrappers are primarily used to produce an accurate stack trace, including passed arguments, without using non-standard browser APIs.

5.3.2 Source code transformations

Source code transformations are based on the idea of modifying the JavaScript code in a way that implicit inaccessible execution state becomes explicit and is not lost upon termination of a function.

A simple implementation would insert JavaScript statements at the *entry* and before all *exits* of every instrumented function. This enables tracing of function calls with arguments and return values in a similar way as the method wrappers described above.

For more complete execution state reification, more source code transformations need to be applied. For example, local variables can be captured by inserting customized tracing code at the right locations. Tracing on a sub-statement level, e.g. tracing the left part of a binary expression, can be done by decomposing these expressions into SSAs. This would also yield information about intermediate computation results but requires sophisticated transformations similar to those done by a full JavaScript compiler. Section 9.1 explores this approach in detail.

5.3.3 Rebuilding Stack

Information about the execution state prior or outside of the interpretation can be valuable during the debugging process.

An example is a user who observes that a function is unintentionally executed multiple times. This user can then set a breakpoint to force interpretation. This will reveal detailed information about the currently executing stack frame but not about the calling functions.

Another example, which was mentioned before, concerns the debugging of unhandled exceptions. If interpretation was not triggered in advance of the exception, then only techniques like method wrappers and source code transformations would be available for the user to find the bug.

Apart from just displaying the calling functions to the user, the execution state outside the interpreter can also be further processed. Traced stack frames with function calls, local variables, and a PC make it possible to rebuild a complete runtime stack which can then serve as input data for the interpreter. This enables the user to resume and step through stack frames even if they were not executed by the interpreter in the first place.

Chapter 6

Implementation

To validate the ideas presented in the previous chapter, we implemented a debugger in Lively that uses partial interpretation combined with method wrappers for execution state reification.

This chapter documents the architecture and relevant design decisions alongside the issues we encountered during the implementation and the solutions we have chosen.

6.1 Restricting JavaScript

The ability to debug JavaScript affects the system in many different ways. The very first issue, which must be taken into account when implementing the Lively debugger, is the language itself. Some features of JavaScript are just very difficult to support, others make execution state reification with partial interpretation impossible. Therefore certain restrictions apply to Lively code to make debugging possible.

6.1.1 Scoped variables

JavaScript's static lexical scoping enables closures accessing state that is otherwise not accessible anymore (see Section 3.1.3). This means that a closure created by the JavaScript VM cannot be executed by the interpreter because it is unable to access the internal variable binding of the JavaScript VM.

Closures are used by Lively in many different places, e.g. to `bind` the pseudo variable `this` to a certain object in a function independently of the actual receiver, to `wrap` a function with another function or to `curry`[1] one or more arguments of a function. Additionally, the polymorphic inheritance of the Lively class system uses closures to bind the `$super` pseudo argument to the parent method in the class hierarchy.

[1]Currying creates and returns a new function which behaves like the original function but one or more of its arguments are bound to certain values.

In all these cases the variable binding does not need to be hidden. Our solution was to change the closures returned by `bind`, `wrap` and `curry` to have an explicit variable mapping that can be accessed by the interpreter. In general, closures can be used in Lively as long as the closure object implements the method `getVarMapping()`; otherwise the interpreter would not be able to debug them.

6.1.2 Other language features

There are also a few other JavaScript language features that have the potential to break the interface between interpreter and native code because they are so powerful.

```
function example1() {
  var obj = {};
  Object.defineProperty(obj, "prop" , {get: function(){debugger}});
  return obj.prop; // will not halt execution
}
function example2() {
  with({a:3}) {
    console.log(a); // "a" is undefined
  }
}
function example3() {
  var a = 3;
  eval("function f(){return a}");
  console.log(f()); // "a" is undefined
}
function example4() {
  return a(); // "a" is undefined
  function a() { return 3; };
}
```

Listing 6.1: JavaScript language features which are not supported by the interpreter.

1. JavaScript objects are usually used as map or dictionaries that store values under a string key. However, it is also possible to define properties with meta-programming, e.g.`Object.defineProperty` [47], which allows custom `get` and `set` functions. The interpreter is not aware of these special properties and will not execute the `get` and `set` functions.

2. There are special scoping rules for JavaScript code inside a `with` block [48], i.e. the lookup for variable names will also consider named properties of the object provided in the `with` head. This *syntactic sugar* would make the scoping im-

plementation of the interpreter much more difficult and its use was therefore discouraged in Lively.

3. One of the most powerful JavaScript features is the `eval` function [49, 50]. It allows arbitrary JavaScript code to get dynamically executed. The impact of the code is hard to foresee which means that the interpreter needs to consider various corner cases in order to implement `eval` without breaking the clean separation between native and interpreted code. The dynamic evaluation of code cannot be completely prevented because Lively depends on it for its browser-based development tools, but it should be used in as few places as possible.

4. Furthermore, the JavaScript scoping rules allow declarations to get implicitly hoisted (see Section 3.1.3), which is often confusing and can easily be avoided by declaring variables prior to using them. Therefore the interpreter does not support implicit declaration hoisting.

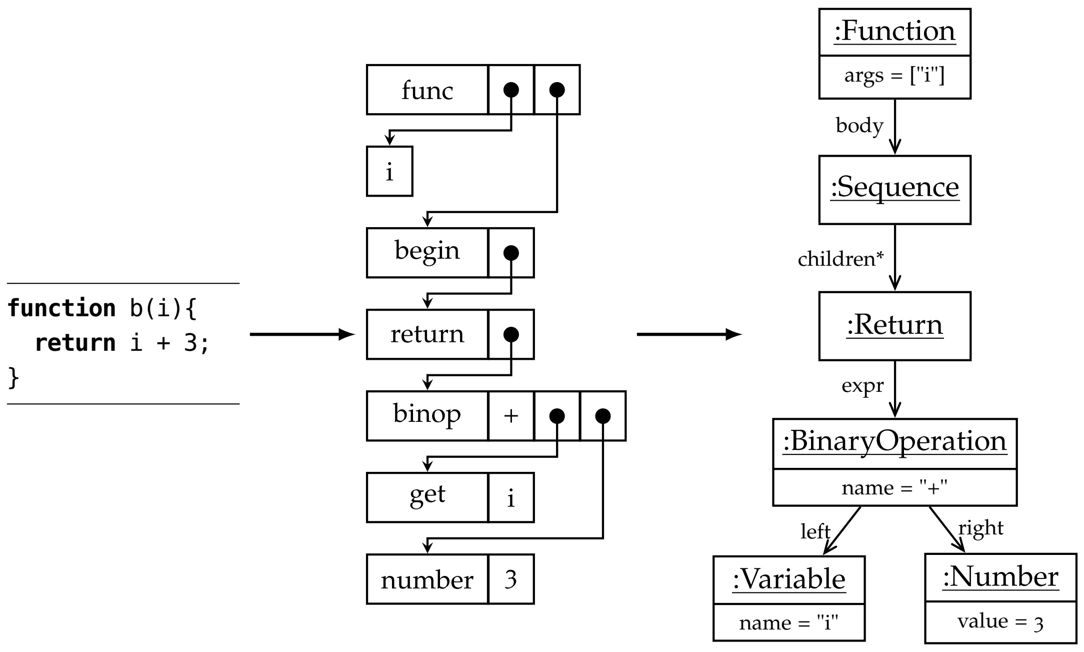
6.2 Parsing

With the language available to the application developer being restricted, the next step is to actually parse the program source and construct an AST suitable for further processing.

The parser itself is written in OMeta [51] which is a pattern matching language based on Parsing Expression Grammars [52]. OMeta was first implemented in Squeak and later rewritten in JavaScript by Warth [53]. The parser creates a tree representation of the source code which consists of nested lists with the first entry of each list begin an identifier for the type of the element. A second OMeta grammar then converts theses lists into an object-oriented AST. This second grammar, the classes of the AST nodes, and a dummy AST visitor are automatically generated from a set of rules in the `SourceGenerator`. Figure 6.1 illustrates this process.

6.2.1 Language extension

The parser adds an additional `debugger` keyword to the JavaScript language. This keyword is an stand-alone statement which does not affect the semantics of preceding and following statements. The purpose of the `debugger` statement is to tell the execution engine to halt the execution at this point and let the user inspect the execution state with a debugger. This way, breakpoints become part of the actual source code which has several disadvantages, e.g. potential unintended sharing of breakpoints between developers and violation of the *separation of concerns* principle [54] due to mixing of



```
function b(i){
  return i + 3;
}
```

Figure 6.1: JavaScript source is parsed to generate nested lists which in turn are transformed to an AST. We use the *list and pointer notation* by Newell and Shaw [2] and the Unified Modeling Language (UML) [3] (UML) *Object Diagram notation* for the AST.

domain code with debugging concerns. However, this way of setting breakpoints is also common in Smalltalk environments[2].

6.3 Interpretation

The core component of the implementation is the interpreter. It evaluates the AST and provides execution state reification.

6.3.1 Evaluating the abstract syntax tree

An interpreter can either use a simple list or a tree as input. The list has advantages for bytecode instructions and branching but we used the AST as input because that way the implementation required less code for most of the operations.

An example AST is shown in Figure 6.1 as result of the parsing process. In general, sequential execution on a tree representation can be done as part of a depth-first traversal. Exceptions to this rule are conditions, which omit branches, and loops, which traverse branches multiple times[3].

The tree traversal is done by using the *visitor pattern* [41] with the interpreter implemented as visitor. As an example, Listing 6.2 shows how the interpreter visits a `BinaryOperation` node like `i + 3`.
Listing 6.2 also shows how abstractions in the JavaScript language are delegated to the underlying JavaScript VM. The `left` node of the binary operation could be a variable with a string value. In that case, the value yielded by `accept` would be a JavaScript string and the plus operator + would automatically convert the second operator, e.g.3, to a string and then concatenate both values. Reusing all these semantics results in a much easier memory management implementation (see Section 5.1.2) and the underlying VM's speed for executing parts of the interpreted program.

A few elements in the JavaScript language are *syntactic sugar* and can be transformed to other elements without loss of generality. Two such examples are given in Table 6.1. Reducing the set of different AST node types by applying these transformation also reduces the implementation size of the interpreter which makes the code easier to maintain and to optimize. This is why most production VMs apply several transformations before handing the resulting AST to an interpreter or compiler. Runtime performance is not a primary concern for our interpreter, therefore it consumes the AST without any transformations.

[2]Squeak provides a `halt` method for all objects which means that statements like `self halt` work as textual breakpoints.

[3]The `for` loop is noteworthy for its update clause which appears prior to the loop body in the source code but is executed after the loop body which inverts the usual AST node execution order

```
lively.ast.Visitor.subclass('lively.ast.InterpreterVisitor',
  'visiting', {
    visitBinaryOp: function(node) {
      var frame = this.currentFrame;
      var leftVal = this.visit(node.left);
      var rightVal = this.visit(node.right);
      switch (node.name) {
        case '+':
          return leftVal + rightVal;
        /* other operators ... */
        default:
          throw new Error('No semantics for binary op ' + node.name)
      }
    }
  } /* other methods ... */
);
```

Listing 6.2: Simplified source code of the `visitBinaryOp` method of the interpreter.

Before Transformation	After Transformation
`return ++i;`	`return (i = i + 1);`
`for (init; condition; update) {` `body;` `}`	`init;` `while (condition) {` `body;` `update;` `}`

Table 6.1: Conversion of special JavaScript abstractions to equivalent abstractions with greater generality.

6.3.2 Scoping

Each function call in the interpreter creates a new stack frame which has its own variable mapping. This mapping is dynamically adjusted as the interpreter encounters new variable declarations. As explained in Section 3.1.3, the pseudo-variable `this` has its own scoping rules, every other variable name is looked up by following the scoping chain. This chain simulates lexical scoping by recursively defining the outer scope of a function to be the surrounding stack frame at the time the function literal is evaluated. The corresponding implementation is shown in Listing 6.3.

Named functions, e.g.`function myfun(){ return 23; }`, are special because the function name will also be added to the variable mapping of the current stack frame. This has practically the same effect as a variable declaration with an anonymous function literal, e.g.`var myfun = function (){ return 23; }`.

Setting the lexical scope of a function to the currently executing stack frame means that, upon termination of the function, its stack frame can still be referenced by functions defined during its execution. In fact, this is exactly how we implemented closures.

```
lively.ast.Visitor.subclass('lively.ast.InterpreterVisitor',
  'visiting', {
    visitFunction: function(node) {
      var frame = this.currentFrame;
      if (node.name) frame.addToMapping(node.name, node);
      node.lexicalScope = frame;
      return node.asFunction();
    },
  } /* other methods ... */
);
```

Listing 6.3: Implementation of closures by retaining the reference to the surrounding scope in the `visitFunction` method of the interpreter.

6.4 Breakpoints

One of the most important features of a debugger is the ability to set *breakpoints* which cause the execution of the program to halt. Breakpoints are always tied to a particular location in the source and trigger right before the statement, line of code or expression will be executed. There are also *watchpoints* and *catchpoints* [11]. Watchpoints trigger when the state of a variable changes to a certain value or a general query regarding the execution state of the program evaluates to a positive result. By combining the query of a watchpoint with a source code location of a breakpoint, a so-called *conditional* *breakpoint* can be created. Catchpoints are similar to watchpoints and trigger when a certain exception is thrown regardless of the source code location.

The implementation of breakpoints in the interpreter was straight-forward because the interpreter can simply stop interpreting at any point during the execution. As described in Section 6.2.1, breakpoints can be set with the `debugger` statement which becomes an AST node visited by the interpreter. The implementation is shown in Listing 6.4.

```
Object.subclass('lively.ast.Visitor',
  'visiting', {
    visitDebugger: function(node) { // do nothing },
  } /* other methods ... */
);
lively.AST.InterpreterVisitor.subclass('lively.AST.ResumingInterpreterVisitor',
  'visiting', {
    visitDebugger: function(node) {
      this.currentFrame.putValue(node, 1); // mark this 'debugger' as visited
                                           // so it will be skipped when resuming
      this.currentFrame.halt(node, true);
    },
  } /* other methods ... */
);
```

Listing 6.4: Visiting explicit breakpoints, i.e.`debugger` statements, in the interpreter.

One major problem arises with our approach of partial interpretation when native functions are on the call stack at the time the execution halts. The JavaScript VM does not provide any means of pausing the execution of these functions, therefore halting the interpretation will return the control flow from the interpreter back to these native functions in either of two ways.
The interpreted code could return normally and use a special return value to signal a premature termination. Unfortunately, this requires the caller of the interpreted code to pay attention to return values which would be a very invasive change since the interpretation mode is triggered dynamically for arbitrary parts of the code.

The other option is to terminate the execution abnormally by throwing a custom exception. This has the advantage that calls will only return a value if the execution successfully terminates. Nevertheless, this is still a leaky abstraction because `try-catch` blocks around the call will then handle the exception. If the code in the exception handler does not differentiate between a real exception and halting the execution, it could unintentionally follow an error protocol, e.g. closing file handles or even retry the call. Additionally, `finally` blocks will get executed regardless of the type of the exception. All these problems need to get solved on a source level which makes the Lively debugger less useful for general purpose JavaScript code.

6.4.1 Stepping

When the execution gets suspended at a breakpoint, the user can inspect the execution state of the program. To observe the changes in execution state over time, the debugger has to provide the means to resume the execution and halt again after a certain interval has passed. This step-by-step debugging, also known as *stepping*, is supported in many state-of-the-art debuggers and also a motivating use case for execution state reification.

The stepping was implemented by having an AST node functioning as a temporary breakpoint, denoted by the `bp` instance variable, which is saved in the stack frame. The interpreter triggers a halt as soon as it encounters or passes the `bp`[4]. This behavior can be seen in Listing 6.7.

```
var a = 23;
var b = f(a + g());
return b;
```

Listing 6.5: Statement stepping halts before assigning `b`

```
var a = 23;
var b = f(a + g());
return b;
```

Listing 6.6: Expression stepping halts before calling `g()`

The size of the step interval is also noteworthy. Compiled languages traditionally use source code lines because there is no one-to-one mapping between AST nodes of the high-level language and the generated code which is often highly optimized. We, however, use the same language for source and target and are therefore free to choose an interval that best suits the user's needs. In most languages one line of code roughly corresponds to one statement. The statement is therefore a familiar unit of stepping for users of other debuggers. Alternatively, expressions could also serve as a more fine-grained unit. The disadvantage of using expressions is unnecessary halting, e.g. when evaluating literals which do not change the observable execution state. The difference between expression stepping and statement stepping is also shown in Listing 6.5 and 6.6.

The current implementation uses statements as primary stepping interval. However, there are a few exceptions, e.g. the condition of an `if` statement, which is an expression but causes the execution to halt when stepping, and the end of a function, which is not a real AST node but denotes the imminent exit and thus also causes the execution to halt when stepping. When a function regularly terminates, the debugger will halt at the next statement after the call in the calling function.

[4]This is necessary in case of a conditional statements because the breakpoint would be set to the `then` branch but the interpreter should also halt if the `else` branch is executed instead.

```
Object.subclass('lively.ast.Interpreter.Frame',
  'resuming' {
    setPC: function(node) {
      // advance the PC of this stack frame to the given node
      // if the node is a sequence, set PC to first statement of sequence
      // otherwise "firstStatement()" just returns the given node
      this.pc = node.firstStatement();
      // halt if the PC is at or past a breakpoint
      if (this.isBreakingAt(node)) this.halt();
    },
    isBreakingAt: function(node) {
      // returns true if the interpreter should halt
      // when evaluating the given node
      if (this.bp === null) return false; // no breakpoint
      if (this.bp === node) return true;  // breakpoint at this node
      return node.isAfter(this.bp);       // execution past the breakpoint
    },
  },
  'stepping', {
    haltAtNextStatement: function() {
      // find next statement right after the current PC
      var nextStmt = this.pc.nextStatement();
      if (nextStmt) {
        // and set a breakpoint at the next statement
        this.bp = nextStmt;
      } else {
        // if there is no statement after the current pc
        // then the last statement has been reached
        // so halt at the function termination
        this.bp = this.getFuncAst();
      }
    },
    stepToNextStatement: function(haltAtCalls) {
      // set breakpoint to next statement
      this.haltAtNextStatement();
      // resume execution
      return this.resume(haltAtCalls);
    },
  }
);
```

Listing 6.7: Implementation of breakpoints and stepping

6.4.2 Descending into function call

The stepping behavior described above is mainly concerned with the source code of a single function. As soon as another function is called, the debugger should either treat the call like all other AST nodes and halt at the next statement after the call, or change the debugging focus to the source code of the called function, halting at the very first statement of the called function. The former **steps over** a call while the latter is **steps into** the call. We implemented this by setting a *meta-breakpoint* which halts every time a function is called regardless of the concrete source code location.

As an interesting consequence of the partial interpretation approach, this *meta-breakpoint* causes the function to always get interpreted. So, stepping into a call interprets it, while stepping over a call uses *shallow interpretation* by default, i.e. it executes the function natively.

6.5 Resuming execution

Given a basic interpreter as outlined in the previous two sections, the next step towards debugging is to enable resuming program execution.

6.5.1 Program counter

There are many different ways to stop the program execution. Resuming, however, is only possible if the saved execution state includes all information necessary to restart the execution at the exact location it stopped.

Many interpreters use a *flat* IR as input, i.e. a non-nested list of primitive IR elements like bytecode instructions. This has the advantage that a numerical index into the list can be used to denote the currently executed element. Branching to another element is done by updating the index to another value; sequential execution simply means to count the index up which explains the term *program counter (PC)*.
Our interpreter has no numerical PC since it uses the AST rather than a one-dimensional IR. The PC is simply the AST node which is currently evaluated by the interpreter. The path from this node to the root of the AST includes all nodes whose evaluation is in progress. Resuming the execution means to resume the evaluation of the PC node and continuing with all the other nodes in this path until finally the evaluation of the root node is completed or the execution halts again.

6.5.2 Resuming with numerical program counter

It would be possible to have a numerical PC by assigning sequential numbers to all AST nodes in a canonical traversal order. These numbers can also be thought of as an index into the postorder linearalized, flat abstract syntax tree. Figure 6.2 shows an AST and its corresponding numbering scheme.

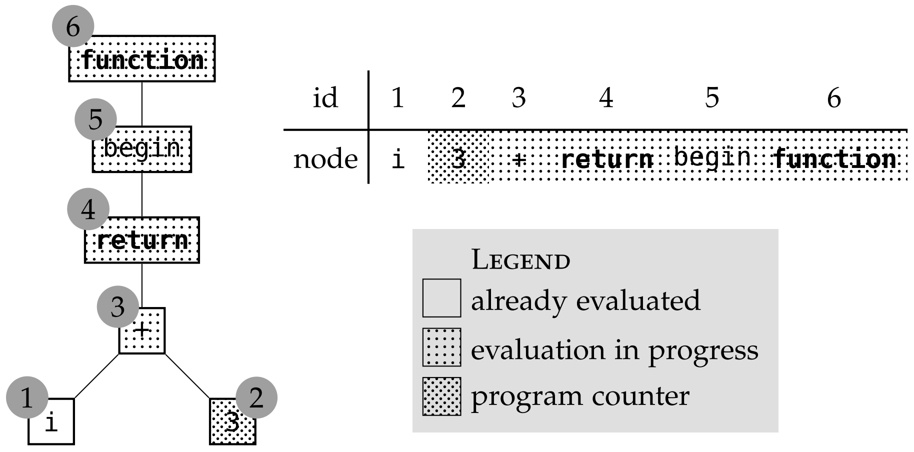


Figure 6.2: Depth-first postorder AST linearization of the code shown in 6.1.

For a given program counter *pc*, all nodes with $id < pc$ have already been evaluated, all nodes with $id > pc$ are either currently in execution or going to be, and the node with $id = pc$ is the node at which the execution should resume.

The implementation is outlined in Listing 6.8. The simple interpreter without resuming capabilities was subclassed to create a resuming interpreter with a slightly different traversal algorithm. If the method `wantsInterpretation()` of the current stack frame returns `true` for the visited node, it will be evaluated, otherwise the evaluation of the node will be skipped.

The normal execution does not need to keep track of the *pc*, so it is set to `null`. This causes line 11 in Listing 6.8 to return `true` which in turn results in normal evaluation of the node.

If the execution is suspended, the *pc* holds the index to the node which caused the execution to halt. An example of this situation can be seen in Figure 6.2 which depicts the suspended execution at $pc = 2$. Upon resuming the execution the tree traversal will be restarted from the root of the AST and the index of each visited node will be compared to the *pc* (see Listing 6.8, lines 10 to 20). The interpreter will skip nodes with smaller indices, e.g. the variable `i` in the example shown in Figure 6.2), and evaluate nodes with greater indices, e.g. the return statement in the example, until the interpreter recursively descends to the AST node with $id = pc$ at which point the normal execution resumes.

The problem of this approach is that intermediate computation results are stored on the stack of the JavaScript VM during the tree traversal. This behavior can be seen in Listing 6.2. The value returned by the evaluation of the left branch is stored in a temporary variable, which resides in the JavaScript VM's stack, at the time the right branch is evaluated. If a node in the right branch causes the execution to halt, the value stored in the temporary variable will be lost. It is still possible to resume the

```
Object.subclass('lively.AST.Interpreter.Frame',
  'resuming', {
    isResuming: function() {
      return this.pc != null;
    },
    resumesNow: function() {
      this.pc = null;
    },
    wantsInterpretation: function(node) {
      if (!this.isResuming()) {
        return true; // normal execution mode
      }
      var nodeIdx = node.astIndex();
      if (nodeIdx < this.pc) {
        return false; // this node was already evaluated
      }
      if (nodeIdx === this.pc) {
        this.resumesNow(); // switch to normal execution mode
      }
      return true; // evaluate this node
                   // (it is either the PC or on the AST path to the PC)
    },
  } /* other methods ... */
);
lively.AST.InterpreterVisitor.subclass('lively.AST.ResumingInterpreterVisitor',
  'visiting', {
    visit: function($super, node) {
      if (this.currentFrame.wantsInterpretation(node)) {
        // call 'visit' of the non-resuming interpreter
        return $super(node);
      }
      return true; // do not visit this AST node and return default value
    },
  }
);
```

Listing 6.8: Resuming implementation based on a numerical PC.

execution by following the algorithm described above which will skip the evaluation of the left branch since it was already computed. However, skipping the evaluation means to return `true`, as shown in line 32 of Listing 6.8, which the interpreter treats as the evaluation result of the left branch. The actual binary operation will most likely return a different result compared to the uninterrupted execution because the original value for the left branch is lost. Recomputing the left branch is also not an option because its evaluation might cause side effects which will then be unintentionally repeated.

6.5.3 Storing intermediate results

We solved the problem mentioned above by storing the return value of each visited AST node in the stack frame. This causes the *garbage collector* to deal with a lot more references, but the memory usage is only slightly increased since no new objects are created.

A plain JavaScript object serves as *dictionary* for storing the computed values. Unfortunately, only strings can be used as *dictionary keys* in JavaScript so the computed values are stored under the string representation of the node's location in the source code of the function[5].

```
Object.subclass('lively.AST.Interpreter.Frame',
  'resuming', {
    getValue: function(node)  { return this.values[node.position()]; },
    putValue: function(node,v){ return this.values[node.position()] = {val: v}; }
  }  /* other methods ... */
);
lively.AST.InterpreterVisitor.subclass('lively.AST.ResumingInterpreterVisitor',
  'visiting', {
    visit: function($super, node) {
      var value = this.currentFrame.getValue(node);
      if (!value) {
         value = this.currentFrame.putValue(node, $super(node));
      }
      return value.val;
    }
  }
);
```

Listing 6.9: Resuming implementation based on stored intermediate results.

If there is a stored value for an AST node, it was already evaluated and will be skipped, otherwise it will be normally evaluated and the return value stored. The im-

[5]For example the number literal AST node 3 in the expression `i + 3` would have the key `"4-5"`

plementation of this approach is shown in Listing 6.9. Because the evaluation can yield arbitrary values, including `null`, the stored values need to be boxed[6]. Additionally, in order to support loops which execute parts of the AST more than once, the computed values of node in the loop bodies need to get removed after each complete iteration.

[6]Alternatively, the function `hasOwnProperty` could be used to differentiate between stored `undefined` literals and *cache misses*.

In contrast to the previous approach there is no difference anymore between normal execution and resuming a suspended function. Therefore it is unnecessary to maintain a PC during execution.

An important drawback of this approach is that it requires significantly more information for resuming the execution[7]. Rebuilding the stack from data gathered outside interpretation, as described in Section 5.3.3, is therefore harder to implement. Only transformations that expose intermediate results, e.g. compilation to SSAs, can provide all the details the interpreter needs to resume the execution (see Section 9.1).

6.5.4 Interpreter call stack

The previous sections mainly dealt with resuming execution of a single stack frame. In order to implement a tracing debugger for Lively, the interpreter also needs to be able to return from function calls when resuming. This is different from resuming during its normal execution because the interpreter can take advantage of the native stack of the JavaScript VM for function calls and returns as long as the execution was not halted.

```
function a() { b(); }
function b() { c(); debugger; }
function c() { debugger; }
```

Listing 6.10: JavaScript functions calling each other. The `debugger` statement in `b()` causes it to get executed by the interpreter.

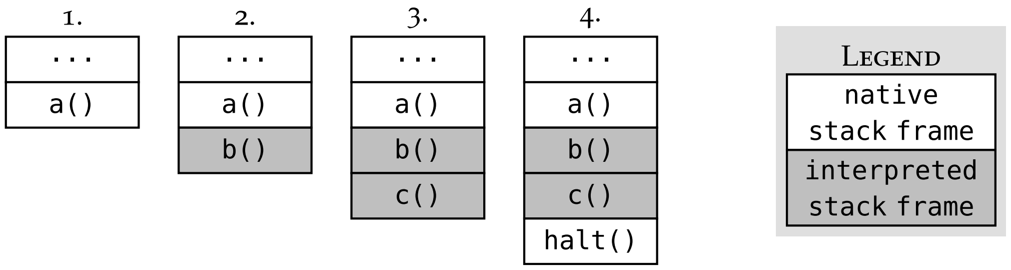


Figure 6.3: Combined native and interpreter call stack at different points in time during the execution of Listing 6.10.

Figure 6.3 shows the relationship between the runtime stack of the JavaScript VM and the interpreter. As the function call `b()` is evaluated, the interpreter will create a new interpreter stack frame for `b` and evaluate the AST of `b` in a new stack frame on the JavaScript VM. If the function terminates normally, its value will be returned by the `visit` function and is immediately available to the interpreter. This causes the

[7]For the approach described in Section 6.5.2 the numerical PC was sufficient.

evaluation of the binary operation in Listing 6.2 to work as expected even if the left branch is actually a function call.

This behavior changes as soon as the execution halts during the evaluation of a call. In fact, we implemented the `halt()` function in a way that all the currently interpreted functions will be suspended. Resuming a complete call stack can now be done in two different ways: either starting at the top-most or at the bottom-most stack frame.

In Figure 6.3, resuming at the bottom would restart `b()`[8] which in turn recursively descends to the stack frame of the **callee**, `c()`, and resumes its execution. When `c()` terminates, the execution continues with `b()` as if the evaluation of `c()` had never been interrupted. This process is shown in Figure 6.4

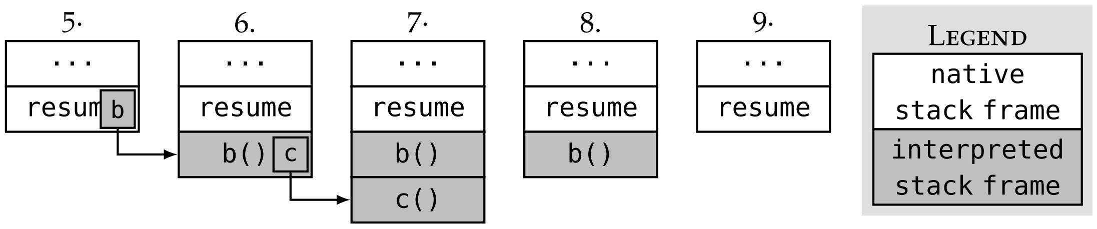


Figure 6.4: Combined native and interpreter call stack at different points in time after resuming the execution in Figure 6.3 following a bottom-up approach.

[8]The function `c()` cannot be resumed because it was natively executed.

Another way to resume a call stack is to start with the top of the stack and resume its execution. This is shown in Figure 6.5. When `c()` terminates, the interpreter has to look up the **caller** of `c` to find that `b` is still suspended and needs to be resumed with the return value of `c`.

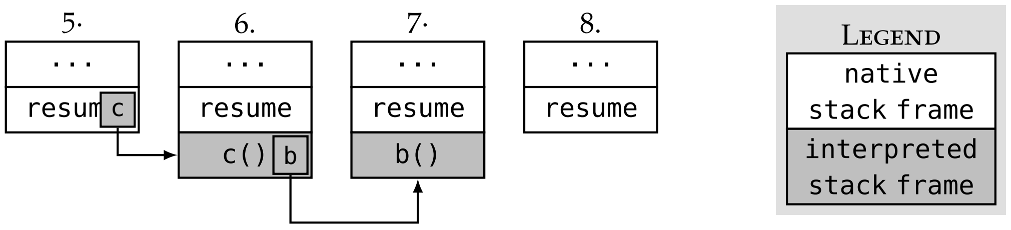


Figure 6.5: Combined native and interpreter call stack at different points in time after resuming the execution in Figure 6.3 following a top-down approach.

We implemented the top-down approach because it optimizes the common case while having a similar complexity as the bottom-up approach. The ability to halt and resume execution is particularly useful for stepping which only concerns the top-most stack frame most of the time. Because of this common use case, we assume that the amount of code executed after a halt is either very small, because there will be a breakpoint immediately after the previous one, or the code will just resume without hitting any further breakpoints. For the latter case, there is no difference in the time spent for resuming execution between these two approaches. For the former case, the bottom-up method always traverses all AST nodes of all stack frames in the call stack, while the top-down method focuses on the top-most stack frame and only considers the rest of the call stack when the function returns. Listing 6.11 shows the implementation.

```
lively.ast.Interpreter.Frame.prototype.resume = function() {
  // resume by traversing the AST and skipping already evaluated nodes
  var result = this.getFuncAst().resume(this);
  // if there is a calling stack frame and it is suspended
  if (this.getCaller() && this.getCaller().isResuming()) {
    // save the return value of this stack frame
    // as evaluation result of the currently executed AST node
    // (the call node in the calling stack frame)
    this.getCaller().putValue(this.getCaller().pc, result);
    // and resume the calling stack frame
    return this.getCaller().resume();
  }
  // either the caller was not interpreted or not suspended
  // so simply return to the next stack frame in the JavaScript VM's stack
  return result; };
```

Listing 6.11: Resuming the interpreter call stack with the top-down approach.

6.6 Native Code

The ability to halt and resume execution is generally limited to code executed by the interpreter. However, there are certain functions that the interpreter cannot execute because they are not implemented in JavaScript. Some of these functions provide an interface to browser APIs, others are part of the language definition and implemented in a compiled language for performance reasons.

```
function foo() {
  var list = [1,2,3];
  debugger; // outer breakpoint BP1
  list.forEach(function(e) {
    alert(e);
    debugger; // inner breakpoint BP2
  });
};
```

Listing 6.12: Array iterator `each` is native but takes a user function as callback.

Listing 6.12 shows an example of a JavaScript code calling a native function. Without the breakpoints BP_1 and BP_2, the function `foo()` will just iterate through the list and output 1,2 and 3. However, enabling these two breakpoints yields different debugging behavior depending on the implementation strategy.

6.6.1 Single interpreter call stack

The simplest possible implementation would be to execute native functions and all functions they call, including user-defined callbacks, without the interpreter. This means that the breakpoint BP_1 causes the execution to halt and additional stepping would just pass over the `forEach` call since there is no way to halt at the start of it and the provided callback will also get executed natively. The inner breakpoint BP_2 will not be evaluated but ignored by the interpreter.

6.6.2 Nested interpreter call stacks

Another approach is to enable debugging for callbacks, etc., by executing them in a new debugging context. This means that the callback is unaware of calling stack frames, e.g. in Listing 6.12 the stack frame for the callback will not know about the stack frame for `foo()` even though it was also interpreted. In the end, this has the effect that both breakpoints BP_1 and BP_2 in the example will work, but using them both together might spawn two debugger instances, one which halts at BP_1 but cannot step into the native function, and one which halts at BP_2 but cannot return to any calling stack frame.

We implemented this approach by wrapping all function literals evaluated by the debugger, which also includes anonymous functions used as callbacks, in a way that calling these functions natively spawns a new interpreter to execute them in a new, independent call stack. If they are called by the interpreter, a normal stack frame will be created instead which becomes part of the existing caller chain.

6.6.3 Unified call stack

It would be possible to extend the behavior described in the previous section to have a single combined call stack even if certain stack frames are native. These native stack frames can still not resume but, nevertheless, the user experience of the debugger improves compared to the two previous approaches. The major obstacle in implementing a unified call stack without additional tracing support is the need to identify the relationship between different call stacks which are separated by native code.

6.6.4 Method wrappers

An alternative strategy for implementing a unified call stack is to instrument the native code in a way that function calls can be traced and this trace can be used as a basis for chaining multiple native and interpreted stack frames. Source code transformations cannot be used because the native functions are not implemented in JavaScript, so we applied method wrappers, which were described in Section 5.3.1, to trace all function calls including those of native functions like `forEach`. Method wrappers are not enabled by default because the instrumentation process is very invasive and takes a lot of time but once they are installed, call stacks also include natively executed stack frames including their arguments and the name as well as the source of the executed function if available.

6.6.5 No native code

A completely different approach is to abolish all native functions except for a few primitives that interact with the environment and do not call user-provided callbacks. This can be done by reimplementing built-in functions in JavaScript. For example, a custom `Array.forEach` can be implemented as shown in Table 6.2.

Surprisingly, the custom `forEach` written in JavaScript actually runs faster than the native implementation. This observation was also made by others who provided custom implementation for `forEach` and `bind` [55]:

> "The point of this test is to show that by creating your own custom `forEach`-lite implementation you can get better performance over native."
>
> John-David Dalton [56]

	Code	Average time per test run	
		Firefox 11.0	Chrome 19.0
Setup	`var values = [];` `for (var i = 0; i < 100; i++){` `values[i] = i;` `}`	–	–
Test	`var sum = 0;` `foreach(values, function(i) {` `sum += i;` `});`	–	–
Native	`function foreach(list, cb) {` `list.forEach(cb);` `}`	$39.7\mu s \pm 0.33\%$	$4.59\mu s \pm 0.42\%$
Custom	`function foreach(list, cb) {` `var l = list.length;` `for (i = 0; i < l; i++) {` `cb(list[i]);` `}` `}`	$3.9\mu s \pm 0.72\%$	$3.84\mu s \pm 0.71\%$

Table 6.2: Performance comparison of native `Array.forEach` and custom `forEach`.

In fact, Andreas Gal from Mozilla mentioned a possible transition from native code to JavaScript in the Firefox browser and predicted a massive speedup for some operations by using JavaScript over C++ [7].

6.7 Lively Integration

In order to make execution state reification and resumable execution actually usable, we implemented a debugger that integrates into the Lively environment. The development was done on a live instance publicly available on the web. Therefore we had to be careful not to affect other users' Lively experience in a negative way, e.g. by introducing a performance penalty or breaking existing code. This was achieved by using COP and grouping all overriding changes to the system in layers. Depending on user preferences, Lively starts with certain debugging layers enabled by default but all these features can also be activated and deactivated on demand.

6.7.1 Debugger statement

As shown in Section 6.4, the interpreter will halt at `debugger` statements. The problem with partial interpretation now is to force interpretation of functions containing these `debugger` statements or else the execution will not halt at these breakpoints.

We have already shown that it is not always possible to determine the debugging scope perfectly. So, instead of reliably triggering the interpretation mode for every function that has a `debugger` statement, we chose to ignore breakpoints from unknown origins, focusing on breakpoints the user set intentionally by writing a `debugger` statement and saving the script or method. This was implemented as a layer overriding the methods `addScript()` and `addCategorizedMethods()` of the Lively class and Morphic system. The function written by the user will be wrapped in the same way as mentioned above in case a traversal of the function AST detected a `Debugger` node.

6.7.2 Debug selection shortcut

In an exploratory programming environment like Lively the user regularly executes snippets of code to change something in the system or to quickly test a few lines of code. We can support the user by additionally providing the ability to observe the execution of this code snippet.

This was implemented by adding an additional keyboard shortcut to all text morphs that triggers the evaluation of the selected code snippets with the interpreter which halts right at the first statement to allow stepping. Text morphs can be tied to a certain context, e.g. the text morphs of the object editor and the object inspector use the object edited or inspected as context. When debugging code typed into these text morphs, the pseudo variable `this` will still point to the original object.

6.7.3 Test runner

Lively also features a *unit testing framework* for test-driven development [57] (TTD)[9]. Tests cannot prove the absence of bugs but they define expected behavior of the code in a way that these expectations can be tested automatically and thereby reveal bugs in the software. Debugging code written with TTD therefore always includes finding the reason a test failed.

In order to support the developer, we enhanced the Lively test runner with the possibility to run the test code with the interpreter. This enables inspection of the call stack at breakpoints, unhandled exceptions, and failed assertions, alongside the ability to step through the test run.

In contrast to the default mode of execution, the debug mode of the test runner triggers interpretation mode for the whole test run. This can be thought of as a *deep*

[9]In fact, the implementation of the interpreter was done with TTD

interpretation as opposed to a *shallow interpretation* which falls back to native execution for all function calls beyond the first. This introduces a significant performance overhead for the test run (see Section 7.3) and should therefore only be used when execution state reification is needed, e.g. for showing a complete call stack after a failed test run.

One unsolved issue is whether to abort the execution of the test suite when the debugger halts or whether to suspend the test suite itself, so it can be resumed later. We implemented the former because the test framework currently not supports asynchronous reporting of test results which would be necessary to suspend a running test suite.

6.7.4 Global error handler

Another obvious use case is to give the user the ability to debug errors, i.e. unhandled exceptions, as soon as they occur.

Without execution stack reification, Lively still defaults to displaying a red box on the screen, summarizing the error message and giving a hard-to-read textual call stack. However, code executed by the interpreter and functions instrumented by method wrappers, attach additional information to the exception which can be used by a global error handler to create a call stack and display it inside a debugger window to the user.

The debugger offers an improved user interface compared to an error message box but its ability to resume the execution is rarely useful. The user can restart a stack frame but that does not answer the question as to what happened right *before* the error occurred [44]. Ideally, the debugger would be able to execute the function backwards in time (see Section 8.2).

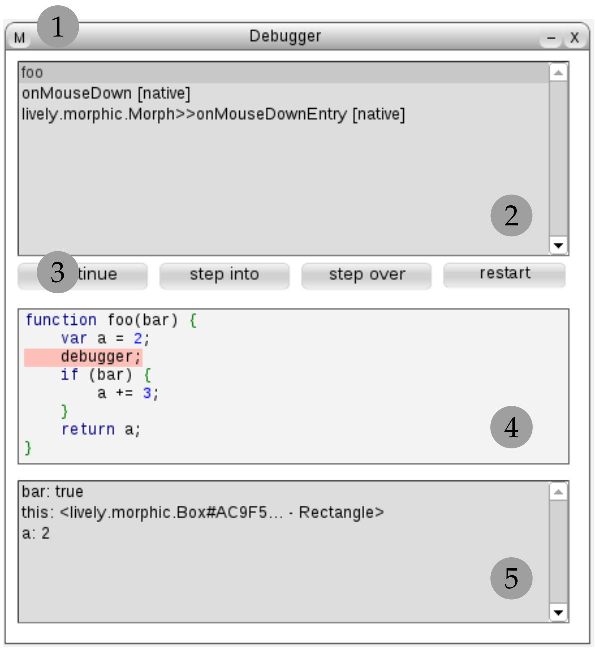
6.7.5 Debugger user interface

Figure 6.6 displays the user interface of the debugger. It is entirely composed of Morphic widgets, therefore it can be stored in the *PartsBin* and loaded on demand into a running Lively environment.

The debugger is a Morphic window with a **title bar** showing the initial event that triggered the debugger, typically an error message or a failed assertion during a test run.

The upper part of the debugger is the **frame list** which displays the call stack. Each line consists of the function or method name, if available, and a special tag for native, not-resumable stack frames. This part is very similar to the traditional call stack viewer in Lively [9].

The **action bar** is a row of buttons for resuming, stepping and restarting the execution of the currently selected stack frame. At the moment, the buttons are simply labeled but future versions of the debugger might provide icons following the *video recorder* metaphor [58] to play, pause, rewind and fast-forward execution.



```
foo
onMouseDown [native]
lively.morphic.Morph>>onMouseDownEntry [native]
```

```
function foo(bar) {
    var a = 2;
    debugger;
    if (bar) {
        a += 3;
    }
    return a;
}
```

```
bar: true
this: <lively.morphic.Box#AC9F5... - Rectangle>
a: 2
```

Figure 6.6: Debugger user interface displaying native and interpreted stack frames

The **source view** displays the JavaScript function corresponding to the current stack frame. It is important to note that this view is unaware of the defining class or module of the function. So, unlike the system code browser, changes to the function's source code cannot be saved. If the stack frame is executed by the interpreter, the current PC is highlighted in red and the user is able to select code snippets and evaluate them by using keyboard shortcuts. This is basically possible with all text morphs in Lively but code evaluated in the source view additionally has access to all variables of the current stack frame which enables the user to inspect and even modify the execution state of the stack frame.

Finally, the **variables list** displays variables bindings of the selected stack frame. If the stack frame was executed by the interpreter, this includes all variables of the stack frame including local variables. If the stack frame was executed natively but instrumented with method wrappers, only arguments passed to the call and the receiver of the method call, `this`, are shown. Primitive values like numbers and strings can be displayed inline but inspecting other variable values, like arrays or complex objects, requires the user to click on the list entry which brings up the Lively object inspector.

Chapter 7

Discussion

This chapter discusses the consequences of partial interpretation and our concrete implementation in Lively on the system as a whole and particularly on the debugging user experience. Additionally, the Lively debugging tools are evaluated in terms of the restrictions they impose on the developer and their runtime performance.

7.1 Restrictions with partial interpretation

The goal of this work was to achieve execution state reification in a way that it can be used to implement a debugger for Lively. We reached this goal by making certain design decisions and compromises that affected the universality of this approach. It would be possible to implement a general purpose debugger by running the complete system on top of the interpreter. Unfortunately, the performance overhead would be unbearable and cripple the Lively user experience, so we built a light-weight on-demand debugger with *partial interpretation* instead. The resulting implementation does not offer execution state reification for all possible JavaScript programs due to its assumptions about the system.

7.1.1 Language restrictions

First of all, certain exotic features of the JavaScript language are not supported because they are incompatible with partial interpretation, difficult to implement, rarely used by the developer, or considered bad style. Some of them will be rejected by the parser, like code blocks used as part of an expression; others are simply not implemented in the interpreter, like the `with` statement.

Additionally, use of the `eval` function, while not completely prohibited, is still discouraged because of its potential to behave differently for interpreted and natively executed code. The same is true for `try`, `catch` and `finally`. They can still be used but the developer needs to be aware that the execution of the `try` block might get suspended by the interpreter which causes an exception to propagate though all native stack frames, incorrectly triggering their `catch` and `finally` blocks.

7.1.2 Explicit variable bindings in closures

The most predominant problem of partial interpretation is the use of closures. A closure that was created by native code and that accesses variables of its outer scope introduces inaccessible execution state for the interpreter.

Suppressing the use of closures is not an option due to their important role in JavaScript language. Our solution was to expose the variable bindings of all closures in a uniform way so that they are recognized by the interpreter. This change needs to get applied to the whole system, even though closures will not cause any problems most of the time. Usually, either both the closure and the surrounding function get executed natively or they are both interpreted. But in the unlikely event that the closure gets interpreted while its outer scope was executed natively, the interpreter has no way to detect the broken variable reference and will either fail to resolve it or in the worst case silently return a shadowed variable as in Listing 7.1.

```
i = 3;                // global variable
function counter() {
  var i = 0;          // private variable
  return function() { // returns the closure
    return ++i        // referencing the private variable
  }
}
var native = counter();
var interpreted = native.forInterpretation();
                      // turns on interpretation mode for the closure
interpreted.call();   // returns '4'
```

Listing 7.1: Inaccessible variable binding in natively executed code results in wrong variable lookup by the interpreter.

7.1.3 System organization

We used method wrappers to enable primitive execution tracing when interpretation is not possible, e.g. when executing native code, or when the interpretation mode was not triggered in advance, e.g. when an unhandled exception occurs.

The method wrapper itself has no requirements regarding the wrapped function, but reliable tracing is only possible when all functions of the system are wrapped. This obviously requires that the system knows about all functions currently in use. In an arbitrary JavaScript program this requirement might be hard to fulfill but in

Lively we have the advantage that the whole codebase is is structured into classes with methods on the one side, and morphs with scripts on the other side. Without this system organization method wrappers are ineffective for this type of tracing.

Method wrappers are not as invasive as source code transformations but they still instrument the system in a way that might affect functions that depend on meta-programming. While invocations of method wrappers are transparent to the caller, the source code of a method wrapper, which can be obtained with the `toString` method, does not represent the source code of the original method.

7.2 Limits of the Lively debugger

The partial interpretation approach and its implementation also influence the functionality of the debugger.

7.2.1 Debugging without interpretation

As a direct consequence of the interpreter approach, the debugger is limited to code written in JavaScript. Built-in functions like `Array.forEach()` cannot be suspended or resumed which impairs the debugging experience, especially if these functions call user-provided callbacks. Section 6.6 proposed possible solutions to this problem.

Another obvious problem is the discrepancy between *first class* interpreted code and *second class* natively executed code. For interpreted code the debugger displays the exact location the execution halts, it shows all local variables, and it supports stepping with the buttons in the action bar. For natively executed code, however, none of these features are available. Even with tracing enabled, only the source code of the called function and the arguments are shown. This may frustrate the user in cases where interpretation cannot be manually controlled, e.g. when debugging unhandled exceptions or when inspecting natively executed stack frames of a suspended function.

7.2.2 Compatibility with meta-programming

Debugging with partial interpretation and method wrappers can also cause conflicts with other applications using meta-programming. Our implementation uses COP, closures and wrappers in a transparent way but certain Lively tools need to be aware of these instrumentations in order to work correctly. For example the Lively serialization mechanism, which is used to store morphs in the *PartsBin*, should serialize the original methods rather than the wrappers created by the debugger to trigger interpretation mode. On another note, changes to the codebase, like new classes or methods added to the system, also need to become instrumented to ensure continuous reliable tracing with method wrappers.

7.2.3 Concurrency

There are also more subtle consequences of the partial interpretation approach like the *pseudo-concurrency* introduced by the ability to halt and resume the execution of a function.

The scope of the concurrency is very limited because it only applies when the debugger halts a function that makes multiple changes to a shared global state. Usually, it is guaranteed that all these changes are committed before the execution of any other code continues. This is not true for the interpreter because it may suspend a computation halfway through its execution. Whether that actually causes a problem depends on the concrete application.

One common issue with this kind of concurrency among others is the handling of user input. In a single-threaded application, halting the user interface thread also halts processing input events. This is important to inspect the execution state without further influencing it. However, the interpreter needs to resume the user interface thread because it is also used for the interface of the Lively debugger itself. A typical example of this issue is drag and drop. Dropping a morph needs to be atomic; either the morph is dropped onto another one or it is still attached to the mouse cursor. When halting code executed during the drop process the morph may be still attached to the cursor. Interacting with the debugger, e.g. by clicking on the *step over* button, will then cause the dragged morph to get dropped on the debugger. A possible solution to this problem is discussed in Section 8.1.1.

7.3 Performance evaluation

Many compilers and VMs support two different modes of execution: development (debug) and production (release) mode. The first focuses on debugging support and the second on runtime performance. Lively is a system targeted at both users and developers, therefore it is important to combine debugging facilities with runtime performance.

We ensured that users who do not use the debugging tools are not affected by loading the code of the debugging tools only on demand. Additionally, users of the debugging tools will not notice a performance overhead as long as they do not trigger interpretation, e.g. by setting breakpoints, or enable tracing. All these features are implemented as COP layers which can be activated and deactivated dynamically.

When all the debugging features like stepping and tracing are used, a decrease in execution speed cannot be prevented. Performance is not a main objective of this thesis, so this section will not go into details and argue about a few percent of increased runtime performance due to missing optimization. Rather this is about the order of magnitude of the implementation's runtime performance. A factor of 1000 makes the difference between a system with an unnoticeable performance overhead and an unusable system.

7.3.1 Interpretation

With the partial interpretation approach only a small portion of the code is executed by the interpreter. Therefore we prioritized simplicity over runtime performance. No profiling was done and even obvious optimizations were omitted, e.g. each function call causes the interpreter to parse the called function's source code due to a missing code cache.

We evaluated the runtime performance of the interpreter with two different benchmarks. The first one executes a single function with a simple summing loop. The exact test code and the measured times are shown in Table 7.1.

As expected, the interpreter executes code considerably slower. In fact, we measured a slowdown factor of about 58 000 (Firefox) / 129 000 (Chrome) compared to native execution.

In addition to the micro-benchmark we also measured the runtime of a typical Morphic method to see how the interpreter performs when executing a more realistic example. This time, multiple nested function calls are involved which means that there are actually two different modes of operation: either function calls beyond the first are executed by the JavaScript VM, i.e. *shallow interpretation*, or nested function calls recursively get interpreted, i.e. *deep interpretation*. The results are shown in Table 7.2).

The purpose of *shallow interpretation* is to limit the performance overhead in cases where interpretation offers no advantage over native execution, e.g. when stepping over a function call in the debugger. In the benchmark it ran 12.4 (Firefox) / 12.8 (Chrome) times faster than *deep interpretation* and 326 (Firefox) / 531 (Chrome) times slower than native.

In conclusion, there is a massive performance overhead when executing code with the interpreter, especially for loops and computationally intensive code. JIT compilers of modern browsers produce very efficient code that runs multiple orders of magnitude faster than an interpreter traversing the tree representation of the program on top of the underlying JavaScript VM. Nevertheless, the gap between native and interpreted execution narrows considerably for more complex user interface code as shown in the second benchmark. Even for *deep interpretation* the slowdown factor decreases to 4 042 (Firefox) / 6 825 (Chrome).

7.3.2 Method wrappers

We also measured the performance overhead introduced by tracing. Tracing was implemented with method wrappers (see Section 6.6.4), therefore it has no effect on the micro-benchmark in Table 7.1. However, tracing does effect the performance of the

	Code	Average time per test run	
		Firefox 11.0	Chrome 19.0

Setup	`var test = function() {`
	`  var sum = 0;`
	`  for (var i=0; i < 100; i++){`
	`    sum += i;`
	`  }`
	`  return sum;`
	`};`
		–	–

Native		$723ns \pm 0.25\%$	$796ns \pm 12.1\%$

Interpretation	`test = test`
	`  .forInterpretation();`
		$42.2ms \pm 0.71\%$	$103ms \pm 0.9\%$

Table 7.1: Micro-benchmark with native and interpreted code.

	Code	Average time per test run	
		Firefox 11.0	Chrome 19.0

Setup	`var test = function() {`
	`  $morph("Rectangle")`
	`    .rotateBy(0.1);`
	`};`
		–	–

Native		$143\mu s \pm 2.36\%$	$80\mu s \pm 2.29\%$

Method Wrappers	`lively.Tracing`
	`  .installStackTracers()`
	`  .startGlobalDebugging();`
		$399\mu s \pm 1.2\%$	$221\mu s \pm 0.24\%$

Shallow Interpretation	`test = test`
	`  .forInterpretation();`
		$46.6ms \pm 2.84\%$	$42.5ms \pm 0.36\%$

Deep Interpretation	`DeepInterpretationLayer`
	`  .beGlobal();`
	`test = test`
	`  .forInterpretation();`
		$578ms \pm 4.93\%$	$546ms \pm 3.95\%$

Table 7.2: Morphic benchmark with native, instrumented and interpreted code.

Morphic-benchmark given in Table 7.2. We observed an increase in the runtime by factor of 2.79 (Firefox) / 2.76 (Chrome). This slowdown is noticeable by the user but, in contrast to interpretation, it does not render the system unusable.

In addition to an increased runtime, method wrappers also consume memory and time for installation. Instrumenting all methods in a typical Lively environment with 250 classes and 5 623 methods takes 2 240 *ms* (Firefox) / 949 *ms* (Chrome) which is not an issue if it happens just once but unacceptable for frequent activations and deactivations.

There is still room for improvement for the tracing implementation. In contrast to interpretation, tracing is intended to be used by default. As such, it needs to have a minimal impact on the user experience. Two different ideas for improving the tracing implementation are given in Section 9.3 and 9.1.

Chapter 8

Related work

Debugging plays an important role in software development and the corresponding debugging tools have been in use for decades now. Unsurprisingly, there has been a lot of research about the future directions of debugging tools in general and for debugging for dynamic languages like JavaScript in particular. This chapter points to noteworthy related work done in the context of JavaScript debugging, omniscient debugging and declarative debugging.

8.1 JavaScript debugging

Most of the JavaScript development tools currently in use, like the Firefox plugin *FireBug* which was described in Section 3.3.1, are modeled after existing tools of other programming environments. Two interesting properties of JavaScript that make it an interesting topic for future research in debugging are, firstly, the unusual user interface API, which interacts with the browser by manipulating the DOM, and, secondly, its use as *mobile code* [59] for the web, i.e. programs written in JavaScript are distributed as source code directly to user's browser.

8.1.1 Record and replay

The single-threaded, event-driven execution model of JavaScript also affects the user interface (see Section 7.2.3). A possible solution to this problem is to break the strong binding of the user interface handling and the associate JavaScript code. On the one hand, this is necessary to interact with the web page without unintentionally triggering additional events that could affect the debugging process, e.g. by clicking a button in the action bar of the debugger interface. On the other hand, this also enables injecting events into the system for debugging purposes without actually having to manually repeat the triggering action in the user interface.

Tools for programmatically interacting with the web page are common for unit and integration testing like *Selenium* [60]. Oney and Myers [61] proposed a system called *FireCrystal* that aims at debugging user interactions. It uses the browser-internal API of Firefox to record the user interface events and later replays these events alongside the executed source code. This is especially useful for debugging complex user interactions, like drag and drop, without manually interacting with the web page. A similar solution called *Mugshot* was presented by Mickens et al. [62] which works across multiple browsers and recreates the execution state of a JavaScript application on another machine. All non-deterministic JavaScript events, including `new Date()`, which returns the current time, are recorded on the user machine and replayed on the developer machine in a way that browser-specific debuggers, like FireBug, can be used in conjunction with *Mugshot* to debug the user input events.

8.1.2 End-user development

Due to its use as *mobile code* in web applications JavaScript is an ideal candidate for end-user development. Exploratory programming environments like Lively greatly benefit from the possibility to load and execute source code at runtime without relying on external tools like compilers. The challenge at hand is to provide an intuitive user interface that enables users, who do not have a strong programming background, to create small applications or scripts. Lively is a suitable host for these systems and early prototypes like *Fabrik* [63, 64] already went into the right direction. A different approach to the same problem was presented by Victor [65] who proposes a live programming environment that makes all changes to the code immediately visible either by simultaneously displaying the graphical output or by showing example inputs to a function and their data flow. In this way JavaScript programs can be visually debugged without stepping through the execution.

8.2 Omniscient debugging

The quote in Section 5.2.2 mentioned the fundamental problem that the chain of events leading to an observable failure is not known in advance; neither by the user nor the debugger. An interesting solution to this problem with active research is *omniscient debugging*, i.e. debugging back in time [66, 67]. In contrast to normal debugging, it allows accessing past execution state which influences the runtime performance and memory consumption of the execution depending on the implementation strategy.

8.2.1 Logging

The most straight-forward approach is to keep a change log. Some implementations like the *Omniscient DeBugger* by Lewis [66] or JHyde by Herrmanns [68] store all changes to the execution state including changes to local variables of a stack frame. The advantage is the precise execution state accurately recreated from the log whilst the drawback is the slowdown factor of 10 to 300 and the memory consumption of 100 MB per second [66].

Another possibility is to limit the log to non-deterministic events. This is very similar to the *record and replay* approach mentioned above, but in languages other than JavaScript this non-determinism can also be the result of concurrency. Deterministic replay of a concurrent system requires changes to the default thread scheduling on the VM level, as presented by Christiaens et al. [69]. While having a low memory footprint, this approach requires repeated execution of the program in order to *step back* in the debugging process.

8.2.2 Snapshots

Alternatively, *omniscient debugging* can also be implemented by taking snapshots during the execution of the program. Many different solutions have been implemented ranging from software-transactional memory to *worlds* [70]. Snapshotting is typically done on the VM level to make use of the garbage collector [71] and have full control over the periodicity and accuracy of the snapshots. The overhead in terms of runtime performance and memory consumption varies accordingly and sometimes it can even be adjusted to the needs of the developer, e.g. the *QVM* by Arnold et al. [72] allows the user to assign a performance budget to the VM that will be spend on debugging capabilities.

8.3 Declarative debugging

A completely different approach is *declarative debugging*. Instead of reifiying the runtime execution state, declarative debugging focuses on distinguishing faulty code from correct code. This can be done on the basis of static analysis, e.g. by detecting reads of uninitialized variables, user queries, e.g. by having a dynamic query-based debugger [73] or using a query language like PQL [74], or slicing which can be done either dynamically or statically and involves manual feedback from developer about the correctness of certain parts of the program [75]. *Declarative Debugging* itself is not closely related to the work on the Lively debugger, but Herrmanns [68] showed that a traditionally tracing debugger can be complemented with declarative debugging features to create a hybrid debugger.

Chapter 9

Future Work

The partial interpretation approach was successfully implemented and can be used to debug Lively programs. This chapter discusses variations and future extensions to partial interpretation which were not implemented yet but which could provide additional features or remove some of the restrictions described in the previous chapter.

9.1 Source code transformations

We implemented tracing of non-interpreted code with method wrappers. Alternatively, source code transformations can be used to instrument the code and enable tracing (see Section 5.3.2). This section shows possible implementation strategies and to what degree they reify the execution state.

9.1.1 Simple call tracing

To achieve the same results as with method wrappers, it would be sufficient to replace all function calls with a trapped version of these calls. If the function call throws an exception then the current source code location, the arguments, and the local variables of the caller's stack frame are saved in the exception object, as shown in Listing 9.3.

```
function f() {
  var a = 1;
  return a + g();
}
```

Listing 9.1: Example JavaScript program

```
function f() {
  var a = 1; // var a = 1
  var _t0 = a;
  var _t1 = g();
  var _t2 = _t0 + _t1;
  return _t2; // return a + g()
}
```

Listing 9.2: Listing 9.1 in SSA form

```
Stack.prototype.addFrame = function(func, astIndex, locals) {...};
function f() {
  var a = 1;
  try {
    var result5_g = g() // pc = 5
  } catch(e) {
    e.stack = e.stack.addFrame(f, 5, {a:a});
    throw e;
  }
  return a + result5_g; // return a + g()
}
```

Listing 9.3: Transformed source code of Listing 9.1 to capture local variables

By doing this, the call stack can be reified without interpretation. However, the stack frame which triggered the exception as well as the intermediate computation results cannot be reified this way which means that the interpreter is not able to resume its execution.

9.1.2 Execution state reification

By applying more sophisticated source code transformations, the execution state can be reified in a way that the interpreter is able to seamlessly take over the execution. This would provide all benefits of interpretation at a dramatically reduced performance penalty.

However, this requires that all intermediate results, e.g. the results of evaluating the two branches of a binary operation, are saved in local variables and that the exact source code location of the currently evaluated expression is known on a sub-statement level.

A straight-forward solution to both of these problems is eliminate expressions in JavaScript by compiling expressions into SSA form prior to their execution on the JavaScript VM as shown in Figure 9.2. SSAs do not have intermediate results and are executed atomically. A numerical PC that increases with each assignment works best in this case because it accurately denotes the currently evaluated part of an expression or statement. Additionally, each assignment is a JavaScript statement and can be wrapped in a `try-catch` block which will be dynamically generated to save all relevant local variables and intermediate results to the current stack frame. An example of this kind of source code transformation is given in Listing 9.4.

```
function addFrame(func, pc, locals,temps) {...};
function f() {
  try {
    var a = 1; // pc = 0
  } catch(e) { e.stack.addFrame(f, 0, {},{}); throw e }
  try {
    var _t0 = a; // pc = 1
  } catch(e) { e.stack.addFrame(f, 1, {a:a}, {}); throw e }
  try {
    var _t1 = g(); // pc = 2
  } catch(e) { e.stack.addFrame(f, 2, {a:a}, {0:_t0}); throw e }
  try {
    var _t2 = _t0 + _t1; // pc = 3
  } catch(e) { e.stack.addFrame(f, 3, {a:a}, {0:_t0,1:_t1}); throw e }
  try {
    return _t2; // pc = 4
  } catch(e) { e.stack.addFrame(f, 4, {a:a}, {0:_t0,1:_t1,2:_t2}); throw e } }
```

Listing 9.4: Transformed Listing 9.2 to reify the complete execution state using SSAs

There are two major drawbacks to this approach. The first is the implementation complexity of the nontrivial transformation from JavaScript expressions to SSAs which is a typical compiler task. And the second is the increase in code size and the corresponding performance overhead.

An alternative to the SSA form is to store intermediate computation results directly within the expressions. This is possible because assignments in JavaScript can be used inside expressions and always return the assigned value. When an exception is thrown, the source code location of the currently executed expression can be detected by analyzing which keys in the _v dictionary were set and which were not. This approach has the advantage that intermediate computation results are stored in the exact same format that the interpreter uses (see Section 6.5.3). Listing 9.5 shows an example of the generated code using this approach.

```
function f() {
  var _v = {};
  try {
    // var a = 1;
    var a = (_v["26-27"] = 1);
    // return a + g();
    return _v["37-44"] = (_v["37-38"] = a) + (_v["41-44"] = g());
  } catch(e) { e.stack.addFrame(f, {a:a}, _v); throw e} }
```

Listing 9.5: Transformed Listing 9.1 to reify the complete execution state by using a value dictionary instead of SSAs

Overall, source code transformations a superior to method wrappers and should replace them in the future to enable execution state reification for non-interpreted code and generally improve the debugging user experience.

9.2 Omniscient debugging

Section 8.2 already mentioned the trend towards *omniscient debugging*. It would be interesting to see how the partial interpretation approach can be applied to omniscient debugging. Runtime performance and memory consumption are blocking a wider adoption of omniscient debugging but both of these problems can be solved by following the same approach as partial interpretation and restricting the scope of debugging to interesting code instead of persisting the complete execution state history.

One possible implementation strategy would be to continuously take snapshots, e.g. at each function call. Stepping back can then be implemented by reverting to the snapshot and restarting execution of the current stack frame until the desired breakpoint right before the last PC is reached. Alternatively, an event log could be held for each *side effect* performed during a function call. When restricted to interpreted function calls, this approach enables the debugger to step back in time with a very limited performance overhead.

9.3 Advances in browser technology

JavaScript currently undergoes a transition from a web scripting language to a general-purpose language for all kinds of applications. Along with this transition, the browser becomes an application platform which has to fulfill the developers' demand for adequate tools and APIs. There are already features in discussion which would make the browser more open for developers, e.g. the proposal for *source maps* [42] which specifically targets dynamically evaluated and compiled code.

As more developer APIs become wide-spread available across different browsers, Lively tools like the debugger can leverage these features to interact with the underlying JavaScript VM. There are existing examples of such APIs, e.g. Python offers the `sys.settracer` function to hook custom code into the interpreter which will be called for each operation, similarly the `java.lang.instrumentation` package in Java offers several ways to monitor the code [76, page 199]. Until these APIs are standardized for JavaScript, we make trade-offs to offer the best possible cross-browser development and debugging experience sacrificing performance and usability.

9.4 Persistent debugging context

The Lively debugger depends on stack frames that were either created by the interpreter or generated during tracing. These stack frames handle arbitrary data at runtime including references to closures or other kinds of functions. This means that there is no easy way to serialize these stack frames and load them at a different point in time or on a different machine.

This breaks the design principle of Lively of having persistent *worlds* which can be personalized, shared and hold their state. When saving a Lively world during a debug session, the debugger window itself will be serialized including all its contents but resuming the execution, stepping or accessing other stack frames is impossible.

In addition to the ability of having debugging sessions lasting longer than the web page lifetime, serialization also allows the live migration of a running JavaScript script across multiple browser instances and multiple machines. This technique is often used in virtualization to keep software system available during hardware failures or upgrades.

Furthermore, a serialized representation of a stack frame could be send to a *web worker* [77] which runs in the background independently of the user interface JavaScript thread and is able to make blocking calls which natively halt its execution. By doing this, the code executed by the debugger is cleanly separated from the rest of the system and cannot interfere with the debugger context and its user interface which would also solve the problem of debugging events while interacting with the browser.

Moreover, stack serialization trivially enables snapshotting as well as record and replay which both can be helpful for debugging user input events and implementing omniscient debugging. With so many applications, persistence for call stacks are definitely an interesting and rewarding topic for future research.

Chapter 10

Conclusions

According to Mikkonen and Taivalsaari [33], "the web is becoming the de facto target platform for advanced software applications". At the same time, development tools for JavaScript, the standard web programming language, are still inferior to the tools available for Java and Smalltalk programmers.

Instead of providing external tools that solve this problem from the outside, Lively is actually part of the web. It is a self-sustaining collaborative web authoring environment in which developers use exploratory programming to continuously change the environment and create new applications and content. The separation of writing, testing and running code blurs in Lively which has the advantage of breaking the tedious *edit-compile-run-debug cycle* of other programming environments. On the other side, programming errors have the potential to break the whole system which makes frequent testing and debugging necessary.

Bugs in Lively are typically fixed by reproducing the faulty behavior, preferably with an automated test case, and then tracing the error back to the code in order to debug it. The Lively debugging tools allow the user to set breakpoints, to inspect the execution state and to step through the code.

The major obstacle in providing this kind of debugging was to reify the execution state of the JavaScript program without relying on the JavaScript VM of the browser. This was achieved by a combination of interpretation and code instrumentation. The code instrumentation gives insight into parts of the execution state without causing a severe performance penalty. The interpreter on the other hand allows unlimited access and manipulation of the execution state at the cost of a 100 000 times slower execution.

By itself, a slowdown factor in that order of magnitude would be unbearable but the partial interpretation approach enables using a different mode of execution for each function call and thereby limits the scope of interpretation to code that benefits from it.

The boundary between native execution and interpretation caused a few issues, especially with closures. These were fixed by restricting the set of supported JavaScript features and relying on the Lively code organization. This also means that our implementation does not solve debugging for general purpose JavaScript code.

In the end, the partial interpretation approach enabled the implementation of a tracing debugger which halts, resumes and steps through JavaScript code without relying on browser-specific APIs. By using source code transformations instead of method wrappers for the code instrumentation in the future, the gap between native and interpreted execution can be closed and seamless on-demand debugging of JavaScript becomes possible.

Bibliography

[1] Adam Ferrari, Alan Batson, Mike Lack, Anita Jones, and David Evans. The x86 assembly guide, 2006. URL http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

[2] Allen Newell and James C. K. Shaw. Programming the logic theory machine. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), pages 230–240, New York, NY, USA, 1957. ACM.

[3] Inc. Object Management Group. Documents associated with uml version 2.4 - beta 2. Technical report, March 2011. URL http://www.omg.org/spec/UML/2.4.

[4] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C Recommendation, December 1999. URL http://www.w3.org/TR/html4.

[5] Steven Pemberton, Daniel Austin, Jonny Axelsson, Tantek Çelik, Doug Dominiak, Herman Elenbaas, Beth Epperson, Masayasu Ishikawa, Shin'ichi Matsui, Shane McCarron, WebGeek Ann Navarro, Subramanian Peruvemba, Rob Relyea, Sebastian Schnitzenbaumer, and Peter Stark. Xhtml 1.0: The extensible hypertext markup language (second edition). W3C Recommendation REC-xhtml1-20020801, World Wide Web Consortium, August 2002. URL http://www.w3.org/TR/2002/REC-xhtml1-20020801.

[6] ECMA International. *Standard ECMA-262*. 1999. URL http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[7] Andreas Gal. High performance javascript, July 2011. URL http://slideshare.net/greenwop/high-performance-javascript. Slides of a talk given at ECOOP 2011, July 28, Lancaster, UK.

[8] Henry Lieberman. The debugging scandal and what to do about it. *Commun. ACM*, 40(4):26–29, April 1997. ISSN 0001-0782.

[9] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The Lively Kernel a self-supporting system on a web page. In Robert

Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 31–50. Springer Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8.

[10] John von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, June 1945.

[11] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9th edition, 2002.

[12] Free Software Foundation Inc. GNU compiler collection, 1987–2007. URL `http://gcc.gnu.org/`. Accessed March 2012.

[13] Free Standards Group. The DWARF debugging standard. URL `http://dwarf.freestandards.org/`.

[14] Sun Microsystems, Inc., Palo Alto, CA. The Java HotSpot performance engine architecture. Technical report, April 1999. URL `http://java.sun.com/products/hotspot/whitepaper.html`.

[15] Mozilla Foundation. MozillaWiki: JaegerMonkey, June 2010. URL `https://wiki.mozilla.org/JaegerMonkey`.

[16] Antero Taivalsaari, James Noble, and Ivan Moore. *Prototype-based programming : concepts, languages, and applications*. Springer, New York, 1999. ISBN 9814021253 9814021253.

[17] Joyent, Inc. node.js. URL `http://nodejs.org/`.

[18] The Apache Software Foundation. Apache Cordova (also known as Apache Callback and PhoneGap). URL `http://incubator.apache.org/callback/`.

[19] The WebKit Authors. JavaScriptCore. URL `http://webkit.org/projects/javascript`.

[20] V8 project authors. Source code of the V8 JavaScript engine. URL `http://code.google.com/p/v8/`.

[21] David Mandelin. Starting JägerMonkey, February 2010. URL `http://blog.mozilla.com/dmandelin/2010/02/26/starting-jagermonkey/`.

[22] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973. ISSN 0001-0782.

[23] Andy Wingo. Value representation in JavaScript implementations, 2011. URL `http://wingolog.org/archives/2011/05/18`.

[24] Nikkei Electronics Asia. Why is the new Google V8 engine so fast?, Jan 2009. URL `http://techon.nikkeibp.co.jp/article/HONSHI/20090106/163615/`.

[25] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer Berlin / Heidelberg, 1991.

[26] David Gudeman. Representing type information in dynamically typed languages. Technical report, University of Arizona at Tucson, 1993.

[27] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[28] The Eclipse Foundation. Eclipse project, 2012. URL `http://eclipse.org/`.

[29] Mozilla Foundation. Firebug, March 2012. URL `http://getfirebug.com/`.

[30] Google. Chrome Developer Tools: Overview, March 2012. URL `http://code.google.com/chrome/devtools/docs/overview.html`.

[31] Mozilla Foundation. Firebug Lite, March 2012. URL `http://getfirebug.com/firebuglite`.

[32] Richard Dale Hoffman. Data processing system and method for debugging a JavaScript program. Patent, 05 2000. URL `http://www.patentlens.net/patentlens/patent/US_6061518/en/`. US 6061518.

[33] Tommi Mikkonen and Antero Taivalsaari. Using JavaScript as a real programming language. Technical report, Mountain View, CA, USA, 2007.

[34] Daniel H. H. Ingalls. The Smalltalk-76 programming system design and implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 9–16, New York, NY, USA, 1978. ACM. doi: http://doi.acm.org/10.1145/512760.512762. URL `http://doi.acm.org/10.1145/512760.512762`.

[35] David Ungar and Randall B. Smith. Self: the power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987. ISSN 0362-1340.

[36] Kevin Dangoor and many CommonJS contributors. CommonJS API Modules/1.0 Specification, March 2011. URL `http://www.commonjs.org/specs/modules/1.0/`.

[37] John H. Maloney and Randall B. Smith. Directness and liveness in the Morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM. ISBN 0-89791-709-X.

[38] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, October 1997. ISSN 0362-1340.

[39] Dean Jackson. Scalable vector graphics (SVG): the world wide web consortium's recommendation for high quality web graphics. In *ACM SIGGRAPH 2002 conference abstracts and applications*, SIGGRAPH '02, pages 319–319, New York, NY, USA, 2002. ACM. ISBN 1-58113-525-4.

[40] Noury Bouraqadi and Serge Stinckwich. Bridging the gap between Morphic visual programming and Smalltalk code. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 101–120, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-084-5. doi: http://doi.acm.org/10.1145/1352678.1352685. URL `http://doi.acm.org/10.1145/1352678.1352685`.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[42] Mozilla Foundation. MozillaWiki: DevTools/Features/SourceMap, January 2012. URL `https://wiki.mozilla.org/DevTools/Features/SourceMap`.

[43] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953. ISSN 00029947. URL `http://www.jstor.org/stable/1990888`.

[44] Henry Lieberman and Christoper Fry. *ZStep 95: A reversible, animated source code stepper*. MIT Press, Cambridge, MA–London, 1998.

[45] Robert Hirschfeld. AspectS - aspect-oriented programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag. ISBN 3-540-00737-7.

[46] Markus Gälli, Adrian Lienhard, and Stéphane Ducasse. The SOM Family, April 2009. URL `http://www.squeaksource.com/ObjectsAsMethodsWrap/`.

[47] Mozilla Foundation. MozillaWiki: Object.defineProperty, March 2012. URL `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty`.

[48] Mozilla Foundation. MozillaWiki: with statement, October 2010. URL `https://developer.mozilla.org/en/JavaScript/Reference/Statements/with`.

[49] Mozilla Foundation. MozillaWiki: eval function, June 2011. URL `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/eval`.

[50] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22654-0.

[51] Alessandro Warth and Ian 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. ISBN 978-1-59593-868-8.

[52] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004. ISSN 0362-1340.

[53] Alessandro Warth. OMeta/JS, January 2012. URL `https://github.com/alexwarth/ometa-js`.

[54] Edsger W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982. URL `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html`.

[55] John-David Dalton. jsPerf: for vs array-foreach, March 2012. URL `http://jsperf.com/bind-vs-custom`.

[56] John-David Dalton. jsPerf: for vs array-foreach, March 2012. URL `http://jsperf.com/for-vs-array-foreach`.

[57] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530.

[58] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1.

[59] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[60] Selenium contributors. Selenium - web browser automation, March 2012. URL `http://seleniumhq.org/`.

[61] Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VLHCC '09, pages 105–108, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4876-0.

[62] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: deterministic capture and replay for Javascript applications. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[63] Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. Lively fabrik a web-based end-user programming environment. In *Proceedings of the 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*, C5 '09, pages 11–19, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3620-0. doi: 10.1109/C5.2009.8. URL `http://dx.doi.org/10.1109/C5.2009.8`.

[64] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: a visual programming environment. *SIGPLAN Not.*, 23(11):176–190, January 1988. ISSN 0362-1340. doi: 10.1145/62084.62100. URL `http://doi.acm.org/10.1145/62084.62100`.

[65] Bret Victor. Inventing on principle, January 2012. URL `http://vimeo.com/36579366`. Talk given at CUSEC 2012, January 20, Montreal, Canada.

[66] Bil Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 7,8, Oct 2003.

[67] Bil Lewis and Mireille Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 96–97, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6. doi: 10.1145/949344.949367. URL `http://doi.acm.org/10.1145/949344.949367`.

[68] Christian Herrmanns. *Entwicklung und Implementierung eines hybriden Debuggers für Java*. Wissenschaftliche Schriften der WWU Münster / 4. Westfälische Wilhelms-Universität, 2011. ISBN 9783840500305. URL `http://books.google.de/books?id=vd3eZwEACAAJ`.

[69] Mark Christiaens, Jong-Deok Choi, Michiel Ronsse, and Koenraad De Bosschere. Record/play in the presence of benign data races. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 3*, PDPTA '02, pages 1200–1206. CSREA Press, 2002. ISBN 1-892512-89-0.

[70] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. Worlds: controlling the scope of side effects. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 179–203, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

[71] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8.

[72] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 143–162, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3.

[73] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, January 2003. ISSN 0928-8910.

[74] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

[75] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

[76] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[77] Neil Graham. Sandbox with web workers, December 2011. URL `http://fingswotidun.com/tests/workerdraw/`.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Berlin, den 21. April 2014
Christopher Schuster