Building Web Based Programming Environments for Functional Programming

by

Danny Yoo

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

May 2012

APPROVED:

Professor Kathi Fisler, Major Thesis Advisor, WPI Computer Science

---

Professor Charles Rich, WPI Computer Science

Professor Joshua Guttman, WPI Computer Science

Professor Shriram Krishnamurthi, Brown University Computer Science

---

Professor Craig Wills, Head of Department

Thesis

JavaScript can be extended to a language that supports sophisticated control operators and satisfies the needs of event-driven functional programs on the Web. An event-driven functional programming model can be supported on both the desktop web browser and mobile smartphones; the asynchronous JavaScript APIs on both platforms can be adapted to provide synchronous interfaces for the end programmer.

Abstract

Functional programming offers an accessible and powerful algebraic model for computing. JavaScript is the language of the ubiquitous Web, but it does not support functional programs well due to its single-threaded, asynchronous nature and lack of rich control flow operators. The purpose of this work is to extend JavaScript to a language environment that satisfies the needs of functional programs on the Web. This extended language environment uses sophisticated control operators to provide an event-driven functional programming model that cooperates with the browser's DOM, along with synchronous access to JavaScript's asynchronous APIs. The results of this work are used toward two projects: (1) a programming environment called WeScheme that runs in the web browser and supports a functional programming curriculum, and (2) a tool-chain called Moby that compiles event-driven functional programs to smartphones, with access to phone-specific features.

Acknowledgements

Let me first start by thanking Guillaume Marceau; from the first day when we met and pair-programmed, I knew I had met an extraordinary soul. It was a privilege to be his hands. He's been a wise mentor and a friend throughout graduate school. He's made my life *interesting*, in the most positive sense of the word.

Alas, a stretch it has been, to be pulled between two worlds. Or, at the very least, between two cities. It could have been straining, but people like Roman Veselinov, Andrey Sklyar, Chris King, Tim Nelson, and Theo Giannakopoulos from WPI's ALAS lab have helped hold me together throughout the whole bewildering process. They fed me pastries, cookies, and other baked goods, which helps. I also want to give thanks to WPI Professors Dan Dougherty and Gary Pollice for their support, stories, advice, and good humor. Furthermore, they fed me turkey and stuffing, which helps too. It has made me glad to say that they made Worcester a home.

My extended stay at Brown University could have been isolating, but it's been anything but that. I have had the privilege to work with many of the brightest and most talented students, all whom helped me with coding, listened, argued passionately, and kept me (mostly) honest. They also helped me feel a little younger. I do not begrudge them for it, not one bit. They include: Zhe Zhang, Brendan Hickey, Ethan Cecchetti, Scott Newman, Will Zimrin, Bill Turtle, and Jeanette Miranda.

Brown's PLT group and friends—Joe Politz, Arjun Guha, Hannah Quay-de la Vallee, Andrew Ferguson, Betsy Hilliard, and Ben Lerner—have been more than welcoming. I'm even a "permaguest", according to the room tag on the door. They, too, have been feeding me well. I'm glad to have been a part of their group.

Dad, Mom, and my brother Richie. What is there to say? I'm afraid I shall have to make the same mistake as Princess Cordelia: there is nothing to compare my love for them, nor words to properly express it. They begot me, bred me, and loved me.

Assuredly, the ideas in this dissertation owe much to Emmanuel Schanzer; his work on Bootstrap to bring ideas from the Ivory Tower to the street has been inspiring. The PLT umbrella research group, and those involved with Program by Design, have been tolerant of my jokes, welcoming of my questions, and forgiving of my mistakes. For that, they have my gratitude. I also thank the other members of my committee: Charles Rich and Joshua Guttman, for their meticulous editing, constructive criticism, and support.

Ultimately, this acknowledgements section needs to talk about my advisors: Kathi Fisler and Shriram Krishnamurthi. I am curious as to how long they will haunt my dreams, but no matter. It was one of Shriram's presentations (The Swine before PERL) that compelled me to chase after graduate school, though he does not know it. Kathi has been wonderfully supportive, and I can only say that I'm a contributing factor to Kathi's declining supply of red editing pens. I thank them both for their heroic efforts to lift my head up to look at the high level, despite my continuing attempts to crane my neck downward toward low-level minutiae. They've been patient, brilliant, generous, and kind. I owe an enormous debt to them both.

Contents

List of Figures

Chapter 1

Introduction

Algebra is a prerequisite for the modern job market, and not only in the science, technology, and engineering fields: almost every professional field uses the language of algebra [22]. Therefore, it's imperative for educational curricula to help students gain a fundamental grasp of algebra, or risk leaving students at a disadvantage for their long-term future. At the middle- and high-school stages, students are studying algebra, coordinate geometry, and simple modeling. Computing and programming can allow algebra to be presented in a more compelling, concrete medium than that of a standard math textbook: rather than use an algebra based exclusively on numbers, computation can allow students to use an algebra on strings, booleans, images, sounds, and graphical animations. In turn, students can eventually realize that mathematics is not just a dry textbook discipline but one that has direct application to topics they care about.

Because of this algebra-rich curricular context, a computational framework aimed for this audience should make heavy use of *functional* programming. In functional programming, programmers write functions in the mathematical sense: a function consumes inputs, produces a value, and is deterministic with no side-effects. It computes the value using algebraic substitution, as taught in schools. Students can use computation to compute answers, but the medium offers more, to be able to build engaging programs that can take full advantage of dynamic interaction.

Traditional functional programming, though, does not provide an ideal model to write interactive programs. A traditional functional program consumes data during initialization, computes a value, and terminates. In contrast, an event-driven program runs continuously and must be able to react to dynamically-generated events such as interactive user input. The World model [9] [10], combines event-driven programming with functional programming. This hybrid model interacts well with a wide variety of platforms, and provides a simplified model for beginners to write useful programs without explicit loops. The combination of a simple computational model (functions and substitution) over rich values (strings, images, etc.) proves to be sufficient for writing quite sophisticated programs, including interactive games.

School teachers who use curricula tied to programming can face two kinds of technical limitations: limited computing power, and locked-down systems. When the systems have few resources, professional software tools fail to run effectively. When systems are locked down, it becomes impossible to install a programming environment. However, a premiere platform has emerged that can run interactive, event-driven programs: the Web. The Web offers compelling benefits to educators and students:

- It is "zero-install", in the sense that any user with a Web browser can access the environment and write programs without installing any other tools. This is an advantage for schools that have very limited computing power or have restrictions on what software can be installed.

- By providing access to Web technologies like the *Document Object Model* (DOM) and *Cascading Style Sheets* (CSS), it can reuse the engineering effort of many companies who are competing to add features and performance. For instance, the addition of sound and video in HTML 5 can be passed on directly to programmers without any changes to the programming language.

- Using the Web's display technology offers another advantage: it gives students an incremental path from Web authoring to programming.

- It can allow for the development of *mashups*, programs composed of several Web applications working together. It offers the promise of deep interoperability with content and programs on the Web, like Flickr and Google Maps, and makes for interesting and novel programming exercises.

- It suggests storing programs in the Cloud, which enables easy global sharing. An important special case of "sharing" is with oneself: students can easily begin work at school, resume at home, continue again at school and so on, always having access to their "files".

Deployment on the Web means deployment to modern computing environments, which include not only desktop computers that are anchored to desks, but laptops and even smartphones. Being able to deploy on mobile devices means students can easily show and share their programs with their peers. Programs that run on these mobile devices can also provide inputs in the form of event-driven sensor data: a game may control the motion of a character by physically tilting the device, for example, or cause some game action to occur based on geographic location.

The use of the web browser as the computing platform lowers the barriers to entry for education. As a concrete example, Bootstrap [37], an educational curriculum for low-income middle- and high-school students, uses the functional event-driven programming model to let students use algebra to build games. A programming environment on the Web allows students to work on a project in the classroom, and continue at home since their work and tools live on the Web.

In short, a programming language that uses algebraic concepts to create event-driven, interactive animations and games has the potential to help students discover the appeal of mathematical concepts and thus make the subject more approachable. Event-driven, functional programs that run on the Web can be made relevant to students because they can be embedded and involved with the outside world, as opposed to being relegated to the classroom. The Web itself offers a flexible, open platform that provides universal access. This dissertation ties these together and shows how to build a functional, event-driven programming environment for the Web.

1.1 Challenges

The Web is replete with examples of language evaluators, online text editors, and networked storage, which makes building a web-based environment seem an exercise in simply putting the pieces together. In fact, modern Web browsers on both the desktop and the smartphone already contain a built-in programming language environment based on the JavaScript programming language. Given JavaScript's omnipresence, it seems natural to build directly on JavaScript.

However, there are challenges with using JavaScript as the base language for a programming environment:

- JavaScript hides runtime errors, with many erroneous expressions evaluating to `undefined` rather than raising exceptions. The lack of visible errors makes it more difficult to trace and correct problems.

- JavaScript has a programming model that does not easily allow computations to be preempted or interrupted, making it difficult to guard the programming environment's behavior against out-of-control computations.

In addition, the educational setting poses additional linguistic requirements that JavaScript fails to satisfy:

- JavaScript doesn't encourage functional programs idioms. JavaScript programs written in functional style can hit runtime limitations with regards to the JavaScript stack ceiling. Furthermore, the asynchronicity of many JavaScript APIs poses difficulties with a functional programming style because values may be produced that are not yet fully initialized.

- JavaScript has poor native support for a numeric model that matches closely with the numeric model that students encounter in school. The only native numeric type in JavaScript is the floating point number, which can have confusing behavior with regards to division and inexact arithmetic.

Furthermore, JavaScript for web browsers poses a particular challenge in terms of its cooperative multitasking and asynchronicity that complicates the implementation of a World programming model on top of it.

Cooperative multitasking and asynchronous control

A key constraint imposed by JavaScript, and one that may surprise non-Web developers, is that it is an inherently cooperative multitasking language that requires programs to be expressed in terms of event handlers that terminate quickly [36]. Programs that don't fit this profile cannot receive input, because control must be yielded to the browser in order to process a browser's input events. Furthermore, a program that holds onto control too long may freeze the rest of the browser, and can even be killed by the browser without warning. This restriction prevents programs written with the assumptions of a synchronous model—functional programs—from running effectively on the Web.

To build a functional, event-driven programming model for the Web is to require a compilation strategy from one high-level programming language with synchronous control structure to one another high level language with an asynchronous one. Although existing compilers, such as Google's GWT [32], perform a translation from high-level languages to JavaScript, they do not explore this need to adapt synchronous to asynchronous control or satisfy the runtime requirements for functional programs. The compiler of this dissertation restricts itself to a subset of the JavaScript language deployed by the majority of web browsers, which prevents the use of some nonstandard features that might otherwise be used to allow programs written in a synchronous style to run.

1.2 Contributions

This dissertation designs, implements, and deploys a functional, event-driven programming environment for the Web. It builds on top of the existing JavaScript evaluator on modern browsers to adapt World programming to the Web, extending it to a language that supports sophisticated control operators that satisfy the needs of event-driven functional programs on the Web. The dissertation deploys this work through two projects:

- WeScheme, a Web-based programming environment for the Scheme [38] and Racket [13] programming languages. WeScheme provides a syntax-highlighting program editor, an interactive tool to run programs on-the-fly, and a hub for sharing programs. WeScheme serves as one of the primary platforms for the Bootstrap curriculum.

- Moby, a toolchain for building smartphone programs that can access phone-specific features through an event-driven functional programming model.

Beneath the surface, both WeScheme and Moby take advantage of the features enabled by the language described in this dissertation: it enables interactive programs to be written in a sequential, synchronous style, a model that is particularly well-suited to beginners for its simplicity. In addition, it provides an event-driven, functional programming framework that supports writing interactive games and applications. Though the underlying virtual machine is very general and supports imperative programming, object-oriented programming, and more [13], many of the technical obstacles are born from the requirement to support the algebraic model on top of the parent JavaScript language environment that does not itself support it.

In order to address these challenges, this work makes the following contributions:

- A mechanism for adapting synchronous interfaces to asynchronous ones, tested by applying it to compiling from Racket to JavaScript.

- A web-based evaluator that supports the needs of a functional event-driven programming environment. It supports features that are not directly provided by JavaScript, such as interrupts, suspending execution, and cooperative timesharing with the browser. It supports recursive processes without being bound by the JavaScript stack, as well as a stack-inspection mechanism that respects tail calls.
- A compiler from a high-level functional language's bytecode to assembly-like JavaScript to improve the performance of a functional language's evaluation in the browser. There are several potential target outputs that a compiler to JavaScript may emit, several of which were explored with in past designs. The compiler produces output to support a high-performance evaluator that doesn't exhaust stack space for procedure calls, and exposes low-level access to an execution's control context.
- An extension of the World event-driven functional programming model to accommodate the DOM and CSS APIs, as well as other external JavaScript APIs.
- A full-featured numeric tower for JavaScript. This tower supports the features of a numeric tower in the Scheme/Racket tradition, including exact integers, rationals, floating points, and bignums.
- A web-based programming environment to exercise and demonstrate these contributions. This environment is self-contained and runs entirely in the browser, and includes the following: an program editor with syntax highlighting and indentation, an interactive evaluator, a break button for interrupting execution, and a sharing feature to publish programs to be viewed and executed on the Web.
- A toolchain for programming smartphones using the same World programming model used in the web-based programming environment. Programmers can access smartphone features through extensions to the World model.

1.3 Overview

The rest of the dissertation describes the design, implementation, and deployment of the programming environment. Section 2 discusses existing work on web-based program evaluators that run on top of JavaScript, and online programming environments.

Section 3 details the functional event-driven programming model. It explains why the asynchronous JavaScript environment can cause difficulties for this model, and shows how control operators can be used to overcome these difficulties. Section 4 extends the World model to interact with features specific to the web browser: the Document Object Model (DOM) and external JavaScript APIs. Section 5 discusses the low-level details of the language compiler, and how the compiler and runtime implement the control operators referenced in Section 3. Section 6 discusses applications of the programming environment to two projects: (1) an online programming environment called WeScheme, and (2) a smartphone application toolchain called Moby. The same functional event-driven programming model in both allows a programmer to write programs that run on these different platforms with minimal change. It also discusses WeScheme's use in the Bootstrap curriculum.

Section 7 evaluates the contributions and discusses future work to improve the performance and scope of the environment.

Chapter 2

Related Work

2.1 JavaScript compilers and evaluators

An interactive evaluator, commonly known as a Read Eval Print Loop (REPL), plays a core role for a programming environment that encourages exploratory programming. REPLs manage the process of compiling and executing programs on-the-fly without interrupting the user with an observable compilation step. On the Web, a REPL can vary in how much of the work of compilation and execution is done on the server versus the client. This work takes a middle-of-the-road approach by compiling on the server-side and taking that result to be run on the client. In the current incarnation of the system, the server performs compilation of source programs so that it can reuse a well-tested, production-level compiler at the front-end.

REPLs such as those on Try Ruby (`tryruby.org`) or Try Haskell (`tryhaskell.org`) take the user's program, evaluate it entirely on the server side, and return the textual output back to the user's browser. This works well for textual output, but is impractical (due to bandwidth concerns) for richer data like images and videos, and obviously useless for interactive applications like games. A server-based REPL has additional problems. These interpreters have a choice of using session state on the server, or re-evaluating the entire sequence of interactions to reconstruct the state of bindings in the REPL. Using session state creates a resource management problem. However, re-evaluating expressions is even more problematic. First, if the definitions are computationally expensive, re-running all the expressions can become intractable. More subtly, re-running computations can produce surprising results. For instance, here is an actual interaction in Try Ruby:

```
>> x = rand(6)
>> x
3
>> x
2
```

That is, the value of `x` appears to have changed between uses even though `x` was not modified! This is because Try Ruby re-evaluates the assignment to `x`, and of course there is no guarantee that it will be bound to the same result. Needless to say, this is a rather confusing interaction. Therefore, this strategy can only be considered useful for toy programs.

In contrast to the strategy of running the program on a server is that of evaluation on the client side. There are several virtual machine interpreters that run inside the browser, such as HotRuby (`hotruby.yukoba.jp`) and OBrowser (`www.pps.jussieu.fr/~canou/obrowser/tutorial/`). These use interpreter implementation techniques similar to those described in Section 5.2.

At the other extreme from running all computation on the server is the idea of performing all computation on the client. The REPL in wscheme (`wscheme.appspot.com`) (not to be confused with WeScheme) lies at the other end of the spectrum, by running entirely inside the user's browser. It accomplishes this by using the Google GWT compiler [32] to compile an existing Java implementation of Scheme [3] into JavaScript.

While this strategy is attractive from the perspective of disconnected computation, wscheme's REPL has a major technical limitation relative to ours: its evaluator does not implement cooperative multitasking (as required by JavaScript), so it is easy to starve the browser of cycles—thus, for instance, wscheme cannot implement a Stop button as found in WeScheme. In addition, its REPL doesn't produce stack traces with errors, and its error messages are not as informative as those of produced by this evaluator.

A few compilers treat JavaScript as if it were a low-level assembly target. Emscripten [44] is one that adapts assembly code to run in JavaScript. It uses a large case/switch form to simulate GOTO jumps, and performs optimizations to replace most GOTOs with the appropriate high-level looping constructs. These optimizations have immediate performance benefits because they avoid the cost of GOTO simulation. Moreover, the loop-introducing optimizations interact virtuously with JavaScript tracing compilers such as TraceMonkey [16], which use loops to trigger JIT compilation. However, these loop optimizations may interfere with separate module compilation, interactive evaluation with REPLs, and cooperative multitasking with the web browser, because the optimizations require a whole-program transformation that is not available in a dynamic code-loading context.

2.2 Web-based programming environments

One of the main applications of this dissertation is WeScheme, which provides an on-line programming environment and a deployment platform that lives entirely on the Web. Much of the related work in this area contains aspects of an environment and deployment platform, though often not in combination.

Lively Fabrik [24] is a programming language that runs entirely inside the browser without plugins. It is a simple, visual programming language consisting of components, pins, and connections with a dataflow semantics; these components are dragged and connected in a visual program editor. Yahoo Pipes (`pipes.yahoo.com`) is another specialized visual programming language for defining RSS feeds from data sources on the Web, using a similar set of tools to connect modules and operations. Each component is implicitly an asynchronous event handler that listens to changes in their inputs. In contrast, WeScheme supports a general-purpose textual language with a strong tie to school mathematics. WeScheme provides synchronous interfaces to the Web's asynchronous programming style, a feature that Lively Fabrik and Yahoo Pipes do not support.

Lively Fabrik runs atop the Lively Kernel [40], which is JavaScript augmented by an implementation of Morphic [28], a Smalltalk GUI interface. In contrast, instead of porting a different GUI library in our work, the existing Web technologies (HTML, CSS) are the user interface platform—thus giving an incremental path from Web page authoring to programming, and also easily incorporating innovations in this rapidly growing field.

CodeMirror (`codemirror.net`) and Mozilla Skywriter (`mozillalabs.com/skywriter/`) (formerly known as Bespin) are both text editor frameworks that work on the Web. Frameworks like these are needed because the plain `textarea` provided by HTML doesn't provide essential support for editing programs. Both these frameworks provide features such as syntax highlighting and indentation, though they use different rendering strategies: Skywriter uses a `canvas` element to render the editor, while CodeMirror uses nested DOM elements.

WeScheme's editor is based on CodeMirror, extended to support the environment's needs. CodeMirror's use of the DOM allows a rich programmatic interface: individual elements can be addressed naturally, for both inspection and manipulation (including, for instance, styling with CSS). Using the DOM element model for a program editor also allows the intriguing possibility of allowing graphical elements to be a used in program source code. WeScheme already allows REPL values to be represented as graphical DOM nodes; it should be technically possible to extend this graphical capability to program source as well, as found in DrRacket.

2.3 Phone environments

Smartphone-development toolchains allow programmers to develop on the desktop and deploy to smartphones. Google App Inventor [31] provides a graphical programming language; it uses the same drag-and-drop metaphors used in the Scratch [27] educational programming environment. Users build programs by dragging and dropping graphical block elements that represent the primitive forms of the language. The language's graphical syntax constrains programs so they are well-formed by construction. It also provides an event-driven model, where specific *when* elements trigger as events occur. Although this environment allows the user to learn about composition by connecting blocks together, its model doesn't encourage the connection between algebraic reduction and computation and therefore misses the educational aims of this dissertation.

2.4 Numerics

Several implementations of bignums exist for JavaScript, such as the jsbn [42] implementation used in our evaluator. On the other hand, although numeric towers have been implemented in many Lisp and Scheme systems, there are fewer examples of well-developed ones for JavaScript. There have been some preliminary efforts to bring a numeric tower to JavaScript [17].

Chapter 3

The Programming Model and its Realization in JavaScript

As motivated in Section 1, a programming language can allow students to explore algebra by writing functional programs. In particular, the language can allow beginners to develop event-driven games with functions. The World model of Felleisen et al. [10] provides such a framework. WeScheme adapts the World model to the Web. This chapter describes the original World model and the main challenges in adapting it to JavaScript.

3.1 The essence of World programming

The World model uses a value called the *world*, which represents the program state in the context of an event-driven computation. Functions can compute new states or on-screen representations based on the arrival of new events. These events can come at regular intervals, like timer ticks, or occur more unpredictably through keyboard or mouse interactions. Unlike traditional event-driven programming, where mutation is necessary to share information across `void`-returning callbacks, the callbacks in World programs are pure functions that consume and produce useful, non-`void` values. The runtime of the World model takes responsibility to hold onto the world between events.

Figure 3.1 shows a simple event-driven animation as an example: a red ball falls down the screen, responding to timer ticks by descending, and finally stops when it hits the floor. The program represents the height of the ball as a single number. Each of the functions (*descend*, *draw*, and *hits-floor?*) consume the world and perform an algebraic computation to produce a value. The *descend* function describes how the ball sinks from one moment to the next. The *draw* function produces an image of the ball that can be drawn on-screen. Finally, the *hits-floor?* predicate describes when the ball has reached the floor.

Because these are pure functions without side-effects, beginners can exercise these functions in the REPL, and even write unit tests to ensure that their functions are producing valid results. Figure 3.1 demonstrates lightweight unit-testing with its use of *check-expect*, which asserts that the value of the first argument matches the expected value in the second argument. The World model enables this lightweight unit testing because all the functions consume and produce plain values, without the need to "set up" and "tear down" to undo the effects of mutation as done in testing frameworks such as JUnit [21].

The following three classes of functions represent the parts of an animation:

- update: how the world changes
- drawing: how the world can be represented on-screen
- (optional) termination: how the computation can stop

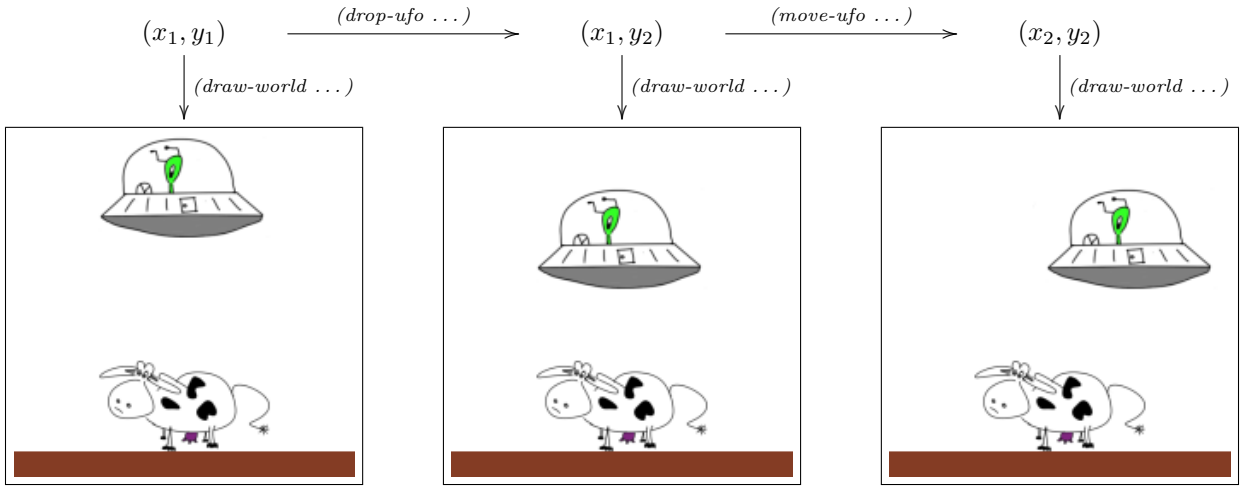
Figure 3.1 An event-driven animation of a ball descending the screen.

```
(define WIDTH 320)   ;; screen width
(define HEIGHT 480)  ;; screen height
(define RADIUS 15)   ;; ball radius

;; The ball is a red circle.
(define RED-BALL (circle RADIUS "solid" "red"))

(define MID-WIDTH (quotient WIDTH 2))
;; The world is the height from the top of the screen.

;; descend: world → world
;; Describes how the world updates in response to time.
(define (descend w) (+ w 5))
(check-expect (descend 5) 10)  ;; a test case

;; hits-floor?: world → boolean
;; Describes when the program should terminate.
(define (hits-floor? w) (> w HEIGHT))

;; Test cases can check that the implementation
;; acts as expected. Violations are treated as runtime
;; errors.
(check-expect (hits-floor? 1000) true)
(check-expect (hits-floor? 0) false)

;; draw: world → scene
;; Describes how the world should be drawn to screen.
(define (draw w)
  (place-image RED-BALL
               MID-WIDTH
               w
               (empty-scene WIDTH HEIGHT)))

;; The use of big-bang starts the world program.
(big-bang 0                      ;; initially, let the height be zero.
          (on-tick descend 1/15)    ;; tick every 15 frames a second
          (to-draw render)          ;; use render to draw the scene
          (stop-when hits-floor?))  ;; and stop when the ball hits the floor.
```

The *big-bang* call in Figure 3.1 begins an event loop which uses the functions provided by *on-tick*, *to-draw*, and *stop-when* to drive its behavior.

3.2 World programs in detail

Two example games demonstrate the interactive features of this programming model: a UFO game and a marble-rolling game. In the first game, a UFO chases after cows. The UFO descends as time passes, and the player can control the horizontal movement of the UFO by pressing the left and right keys.

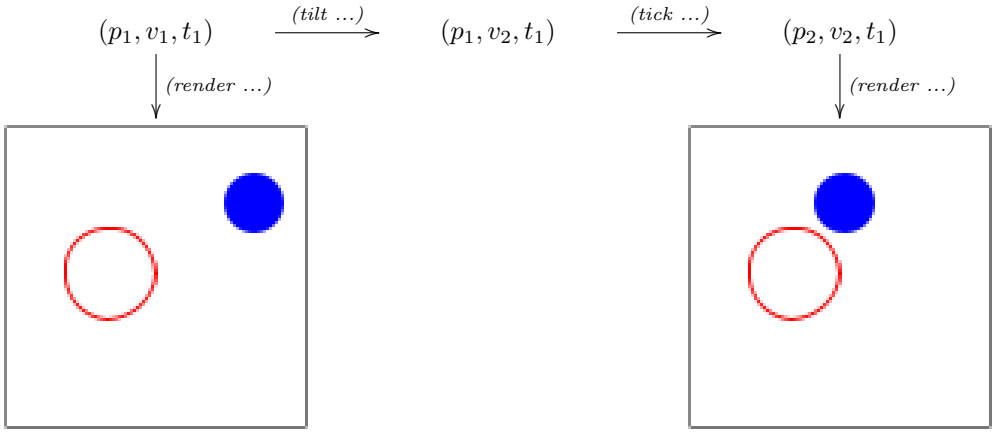


The picture shows a sample trace through the UFO game. The UFO begins at position (x_1, y_1), and moves toward the unsuspecting cow. Like the falling ball example of Section 3.1, a programmer writes functions to describe *update*, *drawing*, and *termination*. The complete source to the UFO game can be seen in Figure 3.2.

The *draw-world* and *ufo-hitting-ground?* functions each consume the world and respectively draws a scene that represents the cow's impending capture and determines the game's end when the UFO reaches the ground. What makes the program interactive is its multiple functions that can update the world: the *drop-ufo* function takes the UFO's position and produces a new position that's closer to the ground. The *move-ufo* function takes both the position as well as a keyboard keystroke to produce the new position.

The programmer provides these functions to the *big-bang* function; from that point forward, the World runtime takes control and starts an event-loop. Throughout the program's extent, the runtime maintains the current position of the UFO. When time ticks by, the runtime calls *drop-ufo* to get the UFO's new position (x_1, y_2), and when the user presses a keystroke, the runtime likewise calls *move-ufo* to compute (x_2, y_2). The runtime also calls *draw-world* periodically to compute the scene, and then displays that scene rapidly enough to show the animation.

The second game serves as a counterpoint to the UFO game. In the marble-rolling game that follows, the player controls the movement of a blue marble toward the red target by pitching and rolling the playing field.



Like the UFO game, the programmer writes functions to manage the world's update in response to events like timer ticks and user input. Compared to the UFO game, the world's information incorporates three pieces of data instead of just one: the position of the player's marble, its velocity, and the position of the target. The functions are otherwise analogous to the first game: the *tick* function moves the position of the player according to the current velocity of the marble and the *tilt* function changes the velocity of the player but leaves its position fixed. The *render* function draws scenes of the player and target. Figure 3.3 shows the source code for this game.

The major difference from the UFO game is its control scheme: the intended driver for *tilt* is meant to be sensor data from accelerometers like the ones in a smartphone. Even with this difference, the model works uniformly on different platforms: it supports both desktop browsers and smartphones, and a phone game that takes tilt input can be easily converted to the desktop simply changing the source of events from tilts back to keystrokes. A minor difference is that it does not terminate.

3.2.1 Implications of the World design

A common element about World programs is that the majority of the functions do not perform side effects.[1] They can be used outside the context of the *big-bang* to be directly unit-tested without the weight of a testing harness.

The World model can be treated as an instantiation of the Model-View-Controller architecture, where the Model is the world value, the View is managed by the drawing function, and the controller is the set of update functions associated to each event type. World's design takes care of several messy details, which can be seen by comparing Figure 3.2 with a version of the program in non-World imperative style in Figure 3.4. Unlike its imperative counterpart, where each callback needs to call *draw!*, the World program delegates the responsibility of redrawing to the framework, as well as the setup and shutdown of event handlers. The World framework does more than just simplify the event-handling boilerplate: it enables the reactive computation to be used in arbitrary expressions.

big-bang as an expression

Although *big-bang* appears only as a top-level statement in Figures 3.1, 3.2, and 3.3, it too can be treated as a plain function that returns a value. In this case, it returns the value of the world upon the event-loop's termination. Allowing *big-bang* to be used as just another expression means that it can be composed, which allows for some interesting uses. The example in Figure 3.5 is inspired by a security dialog box in Mozilla Firefox. In Firefox, when a user installs a new Firefox extension, the browser shows a dialog box to confirm the installation, but the buttons on the box are disabled for a few seconds to make it more difficult for the

[1] The exception is *make-random-posn* in Figure 3.3.

Figure 3.2 The UFO game.

```
(define WIDTH 500)
(define HEIGHT 400)

(define UFO (bitmap/url "http://world.cs.brown.edu/1/clipart/ufo.png"))
(define COW (bitmap/url "http://world.cs.brown.edu/1/clipart/cow-left.png"))
(define GROUND-HEIGHT 15)
(define COW-X (/ WIDTH 2))
(define COW-Y (- HEIGHT (/ (image-height COW) 2) GROUND-HEIGHT))
(define EMPTY-SCENE
  (place-image (nw:rectangle WIDTH GROUND-HEIGHT "solid" "brown")
               0
               (- HEIGHT GROUND-HEIGHT)
               (place-image (nw:rectangle WIDTH HEIGHT "solid" "white")
                            0 0
                            (empty-scene WIDTH HEIGHT))))

(define SCENE-WITH-COW
  (place-image COW COW-X COW-Y EMPTY-SCENE))

;; drop-ufo: world → world
;; Describes how the world updates in response to time.
(define (drop-ufo a-world)
  (make-posn (posn-x a-world) (+ (posn-y a-world) 30)))

;; ufo-hitting-ground?: world → boolean
;; Describes when the program should terminate.
(define (ufo-hitting-ground? a-world)
  (>= (posn-y a-world) HEIGHT))

;; move-ufo: world key → world
;; Describes how the world updates in response to keystrokes.
(define (move-ufo a-world a-key-event)
  (cond [(key=? a-key-event 'left)
         (make-posn (- (posn-x a-world) 10) (posn-y a-world))]
        [(key=? a-key-event 'right)
         (make-posn (+ (posn-x a-world) 10) (posn-y a-world))]
    [else a-world]))

;; draw: world → scene
;; Describes how the world should be drawn to screen.
(define (draw a-world)
  (place-image UFO (posn-x a-world) (posn-y a-world) SCENE-WITH-COW))

(big-bang (make-posn (/ WIDTH 2) 0)  ;; The UFO begins at the top middle,
          (to-draw draw)             ;; using draw,
          (on-tick drop-ufo)         ;; drop-ufo,
          (on-key move-ufo)          ;; move-ufo, and
          (stop-when ufo-hitting-ground?)) ;; ufo-hitting-ground? to drive the event loop.
```

Figure 3.3 Rolling marble game for a smartphone.

```
(define WIDTH 300)
(define HEIGHT 460)
(define RADIUS 30)
(define-struct world (posn vel target-posn))  ;; A world is a posn, a velocity, a target posn.
(define-struct vel (x y))                     ;; A velocity has an x and y component.

(define (tick w) ;; world → world
  (cond
    [(collide? w) (make-world (posn+vel (world-posn w)(world-vel w)) (world-vel w) (make-random-posn))]
    [else (make-world (posn+vel (world-posn w)(world-vel w)) (world-vel w) (world-target-posn w))]))

(define (tilt w azimuth pitch roll)  ;; world number number number → world
  (make-world (world-posn w) (make-vel roll (- pitch)) (world-target-posn w)))

(define (render w) ;; world → scene
  (place-image/posn (circle (+ RADIUS 3) "outline" "red")
                    (world-target-posn w)
                    (place-image/posn (circle RADIUS "solid" "blue")
                                      (world-posn w)
                                      (empty-scene WIDTH HEIGHT))))

(define (collide? w) ;; world → boolean
  (< (distance (world-posn w) (world-target-posn w)) RADIUS))

(define (make-random-posn) ;; → posn
  (make-posn (random WIDTH) (random HEIGHT)))

(define (posn+vel a-posn a-vel) ;; posn velocity → posn
  (make-posn (clamp (+ (posn-x a-posn) (vel-x a-vel)) 0 WIDTH)
             (clamp (+ (posn-y a-posn) (vel-y a-vel)) 0 HEIGHT)))

(define (clamp x a b) ;; number number number → number
  (min (max a x) b))

(define (distance posn-1 posn-2) ;; posn posn → number
  (sqrt (+ (sqr (- (posn-x posn-1) (posn-x posn-2)))
           (sqr (- (posn-y posn-1) (posn-y posn-2))))))

(define (place-image/posn img a-posn a-scene) ;; image posn image → image
  (place-image img (posn-x a-posn) (posn-y a-posn) a-scene))

(big-bang (make-world (make-random-posn) (make-vel 0 0) (make-random-posn)) ;; Random position, zero velocity,
          (on-tick tick)    ;; responding to time vents with updates through tick,
          (on-tilt tilt)    ;; orientation updates with updates through tilt, and
          (to-draw render)) ;; drawing through render.
```

Figure 3.4 The UFO game from Figure 3.2 rewritten in imperative style.

```
;; Assume the existence of the following primitives:
;;
;; draw-to-canvas!: draws an image to the screen
;; add-timer-event-handler!: attaches an event handler on clock ticks
;; clear-timer-event-handler!: removes the clock tick handler
;; add-key-event-handler!: attaches an event handler for keystrokes
;; clear-key-event-handler!: removes the key handler

;; The following abbreviated definitions are the same as in Figure 3.2.
(define WIDTH ...) (define HEIGHT ...) (define UFO ...) (define COW ...)
(define GROUND-HEIGHT ...) (define COW-X ...) (define COW-Y ...)
(define EMPTY-SCENE ...) (define SCENE-WITH-COW ...)
(define (ufo-hitting-ground? world) ...)

;; Global state.
(define ufo-position #f)

(define (drop-ufo!)   ;; void → void
  (set-posn-y! ufo-position (+ (posn-y ufo-position) 30)))

(define (move-ufo! a-key-event)  ;; key → void
  (when (key=? a-key-event 'left)
    (set-posn-x! ufo-position
                 (- (posn-x ufo-position) 10)))
  (when (key=? a-key-event 'right)
    (set-posn-x! ufo-position
                 (+ (posn-x ufo-position) 10))))

(define (draw!) ;; void → void
  (draw-to-canvas! (place-image UFO (posn-x ufo-position) (posn-y ufo-position) SCENE-WITH-COW)))

(define (on-tick!) ;; void → void
  (drop-ufo!) (draw!) (check-stop-when!))

(define (on-key! key) ;; key → void
  (move-ufo! key) (draw!) (check-stop-when!))

(define (check-stop-when!) ;; void → void
  (when (ufo-hitting-ground? ufo-position)
    (clear-event-listeners!)))

(define (clear-event-listeners!) ;; void → void
  (clear-timer-event-handler! on-tick!)
  (clear-key-event-handler! on-key!))

;; Start the event loop:
(set! ufo-position (make-posn (/ WIDTH 2) 0))
(add-timer-event-handler! on-tick!)
(add-key-event-handler! on-key!)
```

user to ignore the dialog message. Figure 3.5 shows an implementation of this kind of interface: it displays a dialog on-screen with two disabled buttons, and after two seconds have passed, the buttons are enabled and the user is allowed to choose a response. Like the previous example, the event-driven computation responds to events like timer and button clicks. *big-bang* is in the test of an **if** expression. Once the user clicks on either the yes or no buttons, the *big-bang* returns, and the caller of *big-bang* receives the final boolean world and continues onward. The ability to treat *big-bang* as an expression opens possibilities for teachers: a teacher may write interactive widgets like Figure 3.5, and students can use these as normal functions. Alternatively, more advanced students may even write their own widgets with *big-bang* and reuse what they know about functions to test their programs.

3.3 Challenges to implementing the programming model

What is surprising, and what motivates this dissertation, is the difficulty of supporting the World programming model in JavaScript on the browser. The main obstacles lie in asynchronicity and event-driven event-loops:

- Asynchronous JavaScript APIs raise an obstacle to functional interfaces.

  The functional model assumes that functions emit usable values on return. However, most Web-based JavaScript APIs present an asynchronous initialization API that notifies when a value is ready to be used: between the start of initialization and notification, the values returned by these APIs are in an undefined, unsafe state.

  For example, one of the functions that should be provided by the World model is *bitmap/url*, which consumes a string URL and produces a bitmapped image of the URL's contents. This function can only be built on top of the native JavaScript API for dynamically loading images. The JavaScript approach to create a dynamic image is to allocate a new Image value, and assign the URL to its `src` attribute. The browser then calls the `Image.onload` callback later to notify when the image has finished loading.

  Because JavaScript image construction uses an asynchronous interface, and although the low-level image constructor produces a value, that value is not safe for use until the image is fully initialized. Any queries on an image's attributes, such as width or height, are ill-defined until then. Only after the asynchronous API signals completion by applying its callback is the value safe for use.[2]

  In the general case, a JavaScript function for an asynchronous API can not act functionally, because there is no built-in mechanism to have a JavaScript function *wait* for the callback to complete.

- JavaScript on the browser can't send or receive events in the middle of normal evaluation.

  In order to receive any events from the browser, such as timer ticks or button presses, the main thread of evaluation needs to be relinquished to the browser. This has consequences for *big-bang*: in order to receive events, *big-bang* needs to give control back to the browser. It must do so by returning to its caller, yet it can not return a useful value to the caller because it hasn't truly finished its computation yet. This violates one of the expected properties from Section 3.2.1, that *big-bang* can be used as a function that returns the world's last value upon termination. In short, the *big-bang* function itself can't act functionally.

  For an online programming environment for the Web, there is another related problem: it should be possible to interrupt a program's execution, to press the *break* button, so that the user can interrupt out-of-control programs. However, if a program is evaluating, that program has control, again preventing any browser user-interface events (like the press of the break button) from being processed until that control has been relinquished to the browser.

In summary, the program's thread of execution must yield to the browser in order to receive new events, but the act of yielding will erase the thread's currently running program context.

[2] In the special case for image loading, if all possible image URLs are known in advance, then those images may be pre-loaded before program evaluation. However, in the general case, URLs are dynamic and no such pre-loading can be performed.

Figure 3.5 Using *big-bang* in expression position.

```
;; yes-no-dialog: string string string → boolean
;; Produces a dialog that displays a message and buttons.
;; The dialog waits two seconds before allowing the user
;; to make a choice.
(define (yes-no-dialog msg yes-text no-text)
  (local [;; The world is either a number, or a boolean.
          ;; It becomes a boolean once the user has made a choice.

          ;; draw-page: world → page
          (define (draw-page w)
            (if (done? w)
                (list (js-div))
                (list (js-div)
                      (list (js-text msg))
                      (list (js-button yes (button-options w))
                            (list (js-text yes-text)))
                      (list (js-button no (button-options w))
                            (list (js-text no-text))))))

          ;; button-options: world → (listof option)
          ;; Enable the buttons if enough time has passed.
          (define (button-option w)
            (if (and (number? w) (< w 2)) '(("disabled" "true")) '()))

          ;; tick: world → world
          ;; If no choice has been made yet, increment the time
          ;; that has passed.
          (define (tick w)
            (if (number? w) (add1 w) w))

          ;; yes: world → world
          (define (yes w) true)

          ;; no: world → world
          (define (no w) false)

          ;; done: world → boolean
          (define (done? w) (boolean? w))]

    ;; The big-bang here produces a boolean once the user makes a choice.
    (big-bang 0
              (to-draw-page draw-page)
              (stop-when done?)
              (on-tick tick 1))))

(if (yes-no-dialog "Sleepy?" "Yeah, I need a nap!" "Nope!")
    (if (yes-no-dialog "Pillow, or books?" "Pass the pillow!" "I want a story!")
        'pillow
        'if-you-give-a-mouse-a-cookie)
    (if (yes-no-dialog "Milk, or cookies?" "Milk!" "Cookies!")
        'cup-of-milk
        'cookies))
```

3.4 Solutions with delimited continuations

In contrast to JavaScript, with its limited operators (function calls, exceptions) to manipulate the control context, some programming languages support a notion of *continuations* to provide mechanisms for non-local control flow. These include the following primitives:

save A save will reify the control context and allow it to be stored somewhere.

prompt A prompt will mark a portion of the control context; this is used in conjunction with aborts to implement linguistic features like exception handling.

abort An abort will erase the current control context and return control back to the outside context, such as the browser.

resume A resume will take a previously-saved control context and resume computation from that point forward.

We adopt these continuation primitives and apply them toward the problems discussed in Section 3.3 as follows:

3.4.1 Asynchronous initialization

Each asynchronous API can be adapted as follows: on an entry into a constructor with asynchronous initialization (such as *bitmap/url*), the runtime saves the current control context. It then assigns a raw callback to resume computation as soon as the value is fully initialized. Finally, it aborts the current computation and gives control back to the browser. As an end result, the adapted function effectively acts as though it were a blocking call in the language, even though it is not truly blocking the browser from performing other computations.

3.4.2 *big-bang*

big-bang is handled similarly to Section 3.4.1, though with a few subtle complications. On an entry into a *big-bang*, the language suspends evaluation by saving the current context and then aborting. The internal event-loop of *big-bang* stores the saved control context, and initializes low-level event handlers. Finally, it aborts back to the browser to allow the JavaScript event-loop to handle events. As raw events are handled, the World implementation calls the functional callbacks to get new worlds. Eventually, when the World framework detects the termination condition, it can take the final world value, restore the control context, and resume the remainder of the computation.

3.4.3 Exception handling

The proper handing of exceptions also poses issues to address. When an exception occurs in a functional callback, the exception should propagate upward, through the event-loop into the original control context. This exception-handling issue also comes up in the context of adapting asynchronous APIs: if a user provides an incorrect URL to *bitmap/url*, the adapter needs to translate such an error back to an exception that is raised in the original calling context. For both situations, the solution is the same: a default exception handler is initialized to catch exceptions or JavaScript errors that reach the top-level. If an exception does occur, the original calling context is reinstated and the exception is thrown upward.

3.4.4 Implementing continuations

Since JavaScript has no direct support for capturing and manipulating the native JavaScript control context, the runtime environment relocates the control context to make it accessible to the language's runtime. The environment adopts a model of delimited continuations [14], which provides the primitives *save*, *prompt*,

abort, and *resume*. Delimited continuations provide a fine-grained model for capturing the control context: they allow portions of the control context to be marked to bound the capture up to a certain point.

This granularity becomes useful when trying to maintain boundaries between subsystems. For example, the interactive REPL of WeScheme may use finalization code that evaluates after each expression, and although that code is on the control context, it should be inaccessible to top-level expressions; an unconstrained capture may allow an expression to repeatedly call the finalization code and break invariants. Likewise, a functional callback should not be allowed to capture the internals of the outermost *big-bang* event-loop. Since the language must deal with both REPLs and callbacks, the encapsulation provided by the delimited continuation model is invaluable.

3.4.5 Virtual machine structure

To implement these operators in a JavaScript context requires the cooperation of a runtime component and its compiler. The runtime holds a reference to a virtual machine (VM), with an explicit array representation for the control stack that is separate from the native JavaScript control stack. An element of this external stack is a JavaScript object whose fields include `label`, `marks`, and `tag` attributes. The runtime assigns JavaScript function values to the `label` attribute; these play the role of return addresses in a low-level machine. The `marks` attribute allows the runtime to attach key/value pairs to the dynamic extent of an evaluation. Finally, the `tag` attribute allows the runtime to annotate the boundaries for continuation capture.

During evaluation, the current continuation can be seen as the currently running JavaScript function plus the elements in the explicit control stack. Because the VM exposes the stack as an accessible value, the runtime can observe and make changes to it. For example, continuation prompts can be implemented by mutating the stack, and continuation capture can be performed by cloning slices of the stack. Within the VM, function calls and recursion can call each other in the usual way. However, when the compiled version of a Racket function returns, it doesn't use the JavaScript `return`, but instead it calls the function on the top of the VM's control stack.

The head of the compiled code for each Racket function decrements a counter in the VM; when the counter goes to zero, the function raises a structured exception value including the function value in its contents. When the runtime sees this class of exception, it extracts the aborted function and the current contents of the control stack, schedules a restart of that function, and finally returns control back to the browser. This allows external JavaScript code to cooperatively multitask with program evaluation and user interface elements to signal new events. Furthermore, it allows external programs to set flags in the VM to signal interrupts to the evaluator. When the scheduled computation restarts, the computation aborts if the VM's break flag has been set. Otherwise, it calls the stored function and restarts the rest of the computation.

Chapter 5 discusses the details of the compiler and its runtime.

Chapter 4

World programming with the DOM: *web-world*

The World programming model described in Chapter 3 provides a Model-View-Controller framework for organizing programs. The View in this original formulation is stateless and exclusive to a single World program; in contrast, the View in a typical web browser page can hold state for each of the elements on a web page, generate events through user interactions, and even be shared across different programs.

The capabilities of the browser-based View affects the structure of World programs that interact with it. This chapter contributes an extension to the World model called *web-world* which allows World programs to use the features of the browser's Document Object Model (DOM). To motivate the extension's design, Section 4.1 first presents an initial prototype library called *js-world* that attempts to use the DOM along the lines of the original World programming model. Although this first attempt shows promise, Section 4.2 describes *js-world*'s weaknesses, all which inform the design of *web-world*. Sections 4.3, 4.4, 4.5, and 4.6 describe *web-world*'s abstractions and functionality. Finally, Section 4.7 presents mechanisms for adapting to external JavaScript APIs to synchronous and asynchronous interfaces, which allow *web-world* programs to take advantage of the capabilities of the Web.

4.1 The first approach: *js-world*

The World programming model allows beginners to write interactive, event-driven applications with pure functions. In order to adapt the World model to the Web, we might take inspiration from existing implementations. We can look at how a World program can generate output by defining a *to-draw* function. A *to-draw* function consumes a world and produces an image to be displayed on screen. Each of these image values can be constructed through image combinators like *overlay*, which overlaps one image atop another, or *bitmap/url*, which constructs a primitive image from a URL. The runtime of the World library takes responsibility to call *to-draw* at appropriate moments, and to render the resulting images to screen.

Unlike a simple windowing output device, a web browser does not present a single bitmapped buffer to render images. Rather, a web browser presents a tree-structured API called the Document Object Model (DOM) whose native data structure is the *DOM tree*. Web programs produce output by mutating the tree, either by adding or removing elements, or changing individual attributes of the tree's nodes. The web browser, in turn, detects these mutations and automatically refreshes its display to match this updated structure.

Given this structured nature of the DOM, we can adapt output to the Web by allowing programmers to generate tree structures, and have the runtime automatically attach these tree structures to the DOM. This approach forms the basis for a prototype library called *js-world*[1], which consists of operators to construct

[1]The *js* prefix stands for JavaScript.

Figure 4.1 A *js-world* program that counts time.

```
;; The world is a number.

;; draw: world → dom
(define (draw w)
  `(p ,(number->string w)))

(big-bang 0
          (on-tick add1 1)
          (to-draw draw))
```

structured DOM values. As a simplification, we can treat s-expressions as a convenient way to construct DOMs[2] with all the power of the basic list operators. At a first glance, this approach appears to be successful. For example, the program in Figure 4.1 shows a counter that uses the *js-world* library; it counts the seconds since a program begins executing.

The original framing of the World model describes a fixed set of events, registered with *big-bang*, that can change the world. One thing that makes the DOM interesting is that it provides a mechanism for both presentation and control, because each element in the DOM can be the source of events that can trigger computation: we can *bind* a function to be called when an event is triggered on a element of the DOM tree. The click of a button and the modification of a text field should be events that can change the world too.

Unlike the setting in the vanilla World model, this set of DOM events is open because browsers continue to embrace new features such as multi-touch events. Therefore, the adaptation of the model should allow the binding of arbitrary event types, and not just a fixed set that contains `"click"`, `"change"`, or `"mousemove"`. We can consider adding a function to the *js-world* API: a function *dom-bind* that enables a World program to connect world-updating functions to the events of the DOM.

$$\textit{dom-bind} : \textit{dom-tree string world-updater} \rightarrow \textit{dom-tree}$$

dom-bind consumes a representation of a DOM tree, an event type, and the world updater to be associated to the particular event type; it captures simple World programs that react to DOM events. For example, the program in Figure 4.2 counts the number of clicks of a button. Here, the `"click"` event does not provide auxiliary information, but other DOM events, such as `"change"` which the DOM triggers when a text field's content is modified, may provide the world-updater the text of the new text content.

4.2 Problems with *js-world*

The initial *js-world* approach appears to be effective when there is no state in the DOM. The button-clicking example in Figure 4.2 begins to hint at a small problem: even if the structure of the DOM has not changed significantly, *draw* is dynamically binding an event handler on every transition of the world. This re-binding is inefficient and, unlike a regular World program, doesn't allow a program to state all the observable events during initialization. Instead, the event-binding in a *js-world* program is non-uniform, where some events are bound in *draw* and others in the call to *big-bang*.

There is a more serious weakness of *js-world*'s adaptation of the World model to the DOM: unlike the inert canvas of World, certain types of DOM elements, specifically form input elements, have state. The HTML5 DOM includes elements such as text fields, sliders, and even calendar date pickers, each of which hold internal values, including the internal position of the cursor selection, the settings of flags, etc.

To understand this problem, consider the program in Figure 4.3, which displays a counter and allows a user to type input into a text field. This program does not provide normal interaction with the user: the user

[2]The implementation of *js-world* opted to provide explicit operators to construct elements, rather than use s-expression syntax, but there were no technical reasons why it could not use s-expressions to express DOM elements.

Figure 4.2 A *js-world* program that counts button presses.

```
;; The world is a number.

;; click: world event → world
(define (click w event)
  (add1 w))

;; draw: world → dom
(define (draw w)
  `(p ,(number->string w)
      ,(dom-bind '(input (@ (type "button")
                            (value "click me")))
                 "click"
                 click)))
(big-bang 0
          (to-draw draw))
```

Figure 4.3 A problematic *js-world* program.

```
;; The world is a number.

;; draw: world → dom
(define (draw w)
  `(div (number->string w)
        (input (@ (type "text")))))

(big-bang 0
          (on-tick add1 1)
          (to-draw draw))
```

Figure 4.4 A revised version of the problematic *js-world* program from Figure 4.3.

```
;; The world consists of both a number and the content of
;; the text field.
(define-struct world (number text))

;; text-change: world event → world
(define (text-change w event)
  (make-world (world-number world)
              (event-value event)))

;; draw: world → dom
(define (draw w)
  `(div (number->string (world-number w))
        ,(dom-bind `(input (@ (type "text")
                              (value ,(world-text w))))
                   "change"
                   text-change)))

(big-bang (make-world 0 "")
          (on-tick add1 1)
          (to-draw draw))
```

may try to type into the text field, but will find that the text abruptly disappears. This happens because as soon as the `on-tick` function has been called, the new DOM tree constructed by *draw* is rendered to the browser; as the world does not store the contents of the text field, *draw* has no functional way to preserve the field.

Since the output of *draw* is a function of the world alone, perhaps the world should also contain the content of the text field. A revised version of the program, Figure 4.4, attempts to correct the problem. However, it too fails spectacularly. Although the user may be able to type into the text field, as soon as an *on-tick* triggers, the text field's cursor jumps back to the beginning of the text field! The cursor's position is also a part of the text field's state, but it has not yet been captured in the world.

This begins to demonstrate the weakness with *js-world*'s approach: even though a *js-world* program may not do anything sophisticated with the DOM, it needs to manage the state of the form elements in the DOM, whether the program cares about the contents or not.

In summary, the approach to treat DOM output as a pure function based only on the world value fails to address several problems:

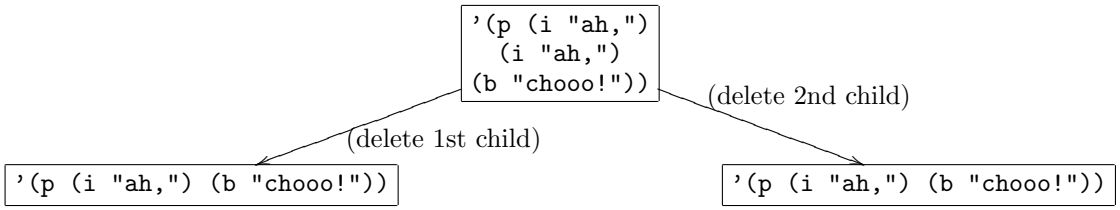
- The rich statefulness of form elements forces *js-world* programs to fully express the state of each form element in a *to-draw*. This approach requires the entire state of each widget to reside in the world.

  The statefulness of the UI can be awkward to express: a text area holds not only the text content, but also the current cursor position or selection. Other HTML form elements can be even richer.

  Furthermore, to force World programmers to manage the state of the DOM is to give them a redundant and tedious task. DOM elements already know how to manage their own state in the browser. The idea that a *js-world* program would repeat the same work as the browser is a cause for concern.

- Normal web interactions depend on the continuity of elements from one world transition to the next. A *to-draw* that constructs fresh DOM values encourages the runtime library to generate output in a discontinuous manner, by deleting the existing elements and repopulating the page with the new elements. If a user is editing a text field, redrawing the world by completely deleting and creating a

Figure 4.5 An trivial example of an ambiguous patch due to tree-diffing.



new tree would offer a poor user experience. For example, the DOM transition can defocus a text field on the web browser.

To maintain the continuity of user elements, the runtime library needs the ability to correlate existing DOM trees on screen with the new DOM values. In *js-world*, this requires a tree-diffing algorithm whose behavior may not even be anticipated, for the patching may be ambiguous. Figure 4.5 shows a simple example, where there are two possible edits that can patch the source tree to the desired result.

- Since programmers must explicit construct a DOM tree from scratch, arbitrary web programs don't nicely compose. This is an especially acute issue in the context of the web because of the presence of external libraries such as Google Maps, which can dynamically inject their own DOM nodes into a page. The implementation of such libraries depend on the identity and persistence of these injected DOM nodes.

  If the *to-draw* of the drawing function can only depend on the local state of the world, then the world is forced to break open the abstractions of private libraries in order to maintain that state across a *to-draw*. To accommodate a library like Google Maps, the world would have to represent the internal state of that library in order to reliably preserve the DOM nodes that the Maps implementation depends on. Not only does this complicate the structure of the world, but it shatters any notion of modularity or composition: as soon as an external library changes its internal DOM representation, the *js-world* program that uses it will need to be changed as well.

4.3 A matter of perspective: including the view

In the framing of the MVC architecture noted in Section 3.2.1, the difficulties introduced by *js-world* can be explained as follows: there is a lack of separation of the state of the View and that of the Model. In *js-world*, these two are conflated. The core insight is to recognize that the state of UI elements in the View should be treated as a peer of the state of the world in the Model. This informs the design of the *web-world* library, which corrects the deficiencies identified in *js-world*. Namely, *web-world*:

1. changes *big-bang* so that it consumes not only the initial world value, but also an initial *view* value. A view provides a functional interface to the DOM. Section 4.4 describes it.

2. relinquishes the majority of the view's state to the browser. World-updating callbacks are adapted to consume not only the world, but also the current state of the view.

3. changes the type of *to-draw* to take in both the world and the view so that output can be expressed differentially in terms of the existing browser's DOM tree.

4. provides a high-level abstraction within the view that presents the DOM as a functional tree structure with localized, context-aware functional update.

The state encapsulated by the DOM is treated independently of the state in the world: each callback that consumes a world now also consumes the *view* value, which allows the callback to inspect the current state of the DOM tree. For example, the types of *on-tick* and *on-key* are changed to the following:

$$\textit{on-tick} : \textit{world} \to \textit{world} \quad \Longrightarrow \quad \textit{on-tick} : \textit{world}\ \boxed{\textit{view}} \to \textit{world}$$
$$\textit{on-key} : \textit{world key} \to \textit{world} \quad \Longrightarrow \quad \textit{on-key} : \textit{world}\ \boxed{\textit{view}}\ \textit{key} \to \textit{world}$$

In general, all world updaters now take the additional *view* input:

$$\textit{world-updater} : \textit{world}\ \boxed{\textit{view}}\ \textit{event-information} \ldots \to \textit{world}$$

This allows world callbacks to functionally query the state of DOM elements in a manner compatible with the functional paradigm. The separation of the view allows the *web-world* library to delegate the maintenance of the view to the web browser and still enable functional access to the view's state when events are processed.

In terms of output, the *to-draw* function includes the existing view as an argument.

$$\textit{to-draw} : \textit{world} \to \textit{DOM} \quad \Longrightarrow \quad \textit{to-draw} : \textit{world}\ \boxed{\textit{view}} \to \boxed{\textit{view}}$$

This leverages one of the conceptual strengths of the original World programming model, which allows the programmer to express changes in the world state in terms of the previous world. The adapted *to-draw* applies the same reasoning to views, in expressing the new view as a function in terms of the previous view.

4.4 Motivation and design of views

Generating a view should be as easy as generating an image. Looking back at the World model, a program generates images through additive operations: its image operations[3] construct new image structures by composing their arguments into larger structures. Pedagogically, the design of these operations resembles the structure of traditional numeric operations, so that programmers may be equipped to understand the operations by applying analogies. One might expect operations on DOM trees to be similar in spirit. Since a view represents the DOM tree, the use of a simple tree representation, such as a s-expression, seems a reasonable choice for views. With the rich support for list construction operations like *cons* and *list*, it seems straightforward to take a similar approach with the DOM.

However, unlike images, the operations on DOM trees are more surgical than additive: typical DOM tree operations dig into a tree's existing structure and make internal updates. With the adaptation of *to-draw* to consume a view that is mostly managed by the browser, it becomes more likely that the programmer will not have prior knowledge about the entire structure of the tree. Without additional API support, manipulating a tree structure requires the use of structural recursion, which can be difficult for beginners to grasp.

For example, we can consider Figure 4.6, which shows a structure representing an enumerated list, embedded in a portion of a DOM tree. We might want to add a new name as an element into the *names* list. With the basic list operations alone, without knowing the outside context in which the substructure has been embedded, it's necessary to pattern match into the tree structure recursively, as shown in Figure 4.7.

Although it may be a worthwhile exercise for students to learn how to write such functions eventually, it's not a task that's appropriate for World programmers. Therefore, it's necessary to provide support for DOM tree operations a higher level than raw list operations.

A view represents an internal reference into a tree that remembers its outer context. The view API, presented in Figure 4.9, provides operations to navigate the context to different areas of the tree. The API encourages navigation on identifier rather than by the raw tree structure. The *view-focus* operator navigates the view's contextual focus to an element with the given *id*. The unfocused portions of the tree, the parent

[3]Examples include: *place-image*, *overlay*, and *beside*.

Figure 4.6 An example of DOM substructure.

```
(define DOM
 '(...
   (p (@ (class "para"))
   "List of dwarves:"
   (ul (@ (id "names"))
       (li "fili") (li "fili") (li "oin") (li "gloin")
       (li "dwalin") (li "balin") (li "bifur") (li "bofur")
       (li "bombur") (li "dori") (li "nori") (li "ori")))
   ...))
```

Figure 4.7 A structural recursive definition for DOM manipulation.

```
;; Add a name to a UL list.
(define (add-name dom name)
  (cond
    [(string? dom)
     dom]
    [(list? dom)
     (cond
       [(names-ul-list? dom)
        `(,(first dom)
          ,(second dom)
          (li ,name)
          ,@(rest (rest dom)))]
       [else
        (cons (first dom)
              (map (lambda (sub-dom)
                     (add-name sub-dom name))
                   (rest dom)))])]))

;; Returns true if the dom tree is the UL with the id "names".
(define (names-ul-list? dom)
  (and (eq? (first dom) 'ul)
       (> (length dom) 1)
       (list? (second dom))
       (eq? (first (second dom)) '@)
       (member '(id "names") (rest (second dom)))))

;; Given a DOM with the subtree structure, we can add a name:
(add-name DOM "balin")
```

Figure 4.8 The code of Figure 4.7 rewritten to use views.

```
;; Add a name to a UL list.
(define (add-name v name)
  (view-prepend-child (view-focus v "names")
                      `(li ,name)))

(add-name DOM "balin")
```

and the immediate sibling subtrees, can still be accessed by using *view-up*, *view-left*, and *view-right*.

$$\begin{aligned} \textit{view-focus} &: \textit{view string} \to \textit{view} \\ \textit{view-up} &: \textit{view} \to \textit{view} \\ \textit{view-left} &: \textit{view} \to \textit{view} \\ \textit{view-right} &: \textit{view} \to \textit{view} \end{aligned}$$

Once the view has been focused on a element of interest, the user can query individual attributes, such as text content by using *view-text* or form element content with *view-form-value*. These functions allow world callbacks to read input from user interactions on the DOM in a purely functional manner, since views are now an argument to the callback.

$$\begin{aligned} \textit{view-text} &: \textit{view} \to \textit{string} \\ \textit{view-form-value} &: \textit{view} \to \textit{string} \end{aligned}$$

The user can also apply a localized, functional update on that element, while the rest of the tree remains unchanged. These operations include *view-update-text*, which can change the text at the focus, and *view-prepend-child*, which can introduce additional structure into the tree.

$$\begin{aligned} \textit{view-update-text} &: \textit{view string} \to \textit{view} \\ \textit{view-prepend-child} &: \textit{view dom-tree} \to \textit{view} \end{aligned}$$

With these elements in place, we can rewrite the example from Figure 4.7 into the significantly shorter form in Figure 4.8.

Since the view can also be a source of events that change the world, *web-world* provides a *view-bind* operation to dynamically bind events in the DOM to world updaters.

$$\textit{view-bind} : \textit{view event-type world-updater} \to \textit{view}$$

The design still permits event handlers to be bound dynamically if necessary. However, since *big-bang* consumes an *initial-view*, the updated API allows the programmer to bind event handlers at the very beginning of the *big-bang* in the common case, restoring the simple structure for event binding present in the original World programming model.

4.5 Examples of *web-world* programs

Programs written with *web-world* are not much more verbose than those in *js-world*. Figure 4.10 shows the counting example (Figure 4.1) rewritten in *web-world*. As in the original program, timer events invoke calls to the *tick* world-updating callback, which computes a new world by adding 1 to the previous world. Since the *tick* callback doesn't need to inspect the DOM, it ignores its view argument. Its *draw* function, on the other hand, uses the view to compute a new view; the runtime preserves the structure and state not mentioned in the update.

Encouragingly, the problem associated with the program from Figure 4.3, where the text field's state disappeared between callbacks, dissolves because the view-updating operations preserve the values in the DOM that have not been explicitly updated. The updated program is in Figure 4.11. Because the *draw* function allows the program to express a differential update, the runtime can easily apply a mutation on the existing browser DOM tree, allowing the state of the text field to be preserved.

Finally, because the browser is given most of the responsibility for managing the view's state, independently of the web-world program, it becomes trivial to query the state of elements in the view without having to pollute the world with extraneous detail. We can create a simple list-manager program that reads

Figure 4.9 View API.

$\rightarrow$*view* : *dom-tree* $\rightarrow$ *view*	Wrap the raw DOM tree into a view.
view? : *any* $\rightarrow$ *boolean*	Check if it is a view.
view-focus? : *any* $\rightarrow$ *boolean*	Check if focus can be moved to *id*.
view-focus : *view string* $\rightarrow$ *view*	Move the focus to the element with the given *id*.
view-left? : *view* $\rightarrow$ *boolean*	Check if focus can be moved to the previous sibling.
view-left : *view* $\rightarrow$ *view*	Move the focus to the previous sibling.
view-right? : *view* $\rightarrow$ *boolean*	Check if focus can be moved to the next sibling.
view-right : *view* $\rightarrow$ *view*	Move the focus to the next sibling.
view-up? : *view* $\rightarrow$ *boolean*	Check if focus can be moved to the parent.
view-up : *view* $\rightarrow$ *view*	Move the focus to the parent.
view-down? : *view* $\rightarrow$ *boolean*	Check if focus can be moved to the first child.
view-down : *view* $\rightarrow$ *view*	Move the focus into the first child.
view-forward? : *view* $\rightarrow$ *boolean*	Check if focus can be moved forward.
view-forward : *view* $\rightarrow$ *view*	Move the focus forward.
view-backward? : *view* $\rightarrow$ *boolean*	Check if focus can be moved backward.
view-backward : *view* $\rightarrow$ *view*	Move the focus backward.
view-text : *view* $\rightarrow$ *string*	Get the text content at focus.
view-update-text : *view string* $\rightarrow$ *view*	Set the text content at focus.
view-bind : *view string world-updater* $\rightarrow$ *view*	Bind a world-updating callback to the event type.
view-show : *view* $\rightarrow$ *view*	Show the focused element.
view-hide : *view* $\rightarrow$ *view*	Hide the focused element.
view-attr : *view string* $\rightarrow$ *string*	Get attribute value at focus.
view-has-attr? : *view string* $\rightarrow$ *boolean*	Check if focus has the named attribute.
view-update-attr : *view string string* $\rightarrow$ *view*	Set attribute at focus.
view-remove-attr : *view string* $\rightarrow$ *view*	Remove attribute at focus.
view-css : *view string* $\rightarrow$ *string*	Get CSS style at focus.
view-update-css : *view string string* $\rightarrow$ *view*	Set CSS style at the focus.
view-id : *view* $\rightarrow$ *string*	Get *id* at focus.
view-form-value : *view* $\rightarrow$ *string*	Get form value at focus.
view-update-form-value : *view string* $\rightarrow$ *string*	Update form value at focus.
view-insert-right : *view dom-tree* $\rightarrow$ *view*	Insert as immediate next sibling.
view-insert-left : *view dom-tree* $\rightarrow$ *view*	Insert as immediate previous sibling.
view-append-child : *view dom-tree* $\rightarrow$ *view*	Insert as last child.
view-prepend-child : *view dom-tree* $\rightarrow$ *view*	Insert as first child.
view-remove : *view* $\rightarrow$ *view*	Remove element at focus.

Figure 4.10 A *web-world* version of the clock-ticking program in Figure 4.1.

```
;; The world is a number.

;; draw: world view → view
(define (draw w v)
  (view-update-text (view-focus v "id") (number->string w)))

;; tick: world view → world
(define (tick w v)
  (add1 w))

(big-bang 0
          (initial-view (->view '(div (@ id "n"))))
          (on-tick tick 1)
          (to-draw draw))
```

Figure 4.11 A *web-world* version of the program in Figure 4.3.

```
;; The world is a number.

;; draw: world view → view
(define (draw w v)
  (view-update-text (view-focus v "id") (number->string w)))

;; tick: world view → world
(define (tick w v)
  (add1 w))

(big-bang 0
          (initial-view (->view '(div (div (@ id "n"))
                                      (input (@ (type "text"))))))
          (on-tick tick)
          (to-draw draw))
```

Figure 4.12 A simple list maker in *web-world*.

```
;; The world is a list of strings.

;; add-item: world view → world
;; Add the text in the textField to our list of strings.
(define (add-item w v)
  (cons (view-text (view-focus v "textField"))
        w))

;; draw: world view → view
;; Render a string representation of the strings into the paragraph.
(define (draw w v)
  (view-update-text (view-focus v "para")
                    (format "~a" w)))

;; view-template: view
(define view-template
  (->view '(div (input (@ (type "text") (id "textField")))
                (input (@ (type "button") (id "addButton")
                          (value "Add!")))
                (p (@ (id "para"))))))

(big-bang (list "milk" "eggs")
          (initial-view
           (view-bind (view-focus view-template "addButton")
                      add-item))
          (to-draw draw))
```

in an item from a text field whenever a button is pressed. This program is shown in Figure 4.12. In this framework, the programmer does not need to manage the state of the text field; *web-world* delegates that effort to the web browser.[4]

4.6 Implementing views

The components of a view allow the runtime to provide a functional API to the DOM tree that can directly express the functional updates as imperative changes on the browser. The full structure of a view consists of three components:

$$\text{view} : \text{tree-zipper(DOM)} \times ((\text{listof DOM}) \to \text{void}) \times \text{nonce}$$

The first component, a tree zipper (discussed in Subsection 4.6.1), enables localized functional editing of the DOM. The second component, the list of functions, represents the mutations that, when replayed on the live browser DOM, result in a tree with the same structure as that in the zipper. The third component, the nonce, is a freshly-generated opaque value that allows the runtime to detect a dependency between an input view and the output of operations on that view.

As an example, to change the text content of an element in a view involves:

- focusing the view on the affected element,

[4]HtDP 2e [11] includes an extended exercise to design a world program that presents a text field (Exercise 2.5.6). It is instructive to see how much work is necessary to manage text field's state.

- adjusting the text content, using the properties of zippers to do the localized edit,
- recording the text-changing operation as a mutation that will perform the change to the DOM imperatively, and
- preserving the nonce.

When *to-draw* is called by the *web-world* runtime, a fresh view is constructed holding a representation of the current DOM tree in the browser, an empty sequence of mutations, and a unique nonce. When the result of *to-draw* is returned to the runtime, then the runtime checks to see whether or not the view shares the same nonce as that of the input. If so, then it knows that there is a direct dependency between the on-screen browser and the view value, and that it can replay the mutative operations on the real browser DOM to replicate the view's structure. This allows only the differences between the old view to be applied mutatively to the browser DOM. Otherwise, then there is a deliberate discontinuity, and the DOM on the browser is discarded and replaced by the content in the zipper.

4.6.1 Zippers

As discussed in Section 4.4, an API that presents a contextual and functional API for trees can be easier for beginners to use than a raw tree API. In imperative languages, popular APIs for the DOM such as JQuery [20] have a concept of a navigatable context for editing within a tree, with methods for moving the context from one portion of the tree to the next. The implementation of the context in these languages is a fairly thin one, because the APIs in these languages don't emphasize functional, algebraic interfaces.

Functional programmers can use a structure called a zipper [18], which provides a mechanism for defining a similar kind of localized context. Zippers take a tree with internal nodes and provide convenient in-place navigation and functional operations of that tree. As the name suggests, a zipper can *open* up a node: this creates a new zipper that contains the immediate child of the node, along with the parent zipper. While a node is open, its attributes can be adjusted in constant time, without having to reconstruct the whole tree. Navigating to a node's immediate next or previous sibling can also be done in constant time.

One of the side benefits of a zipper is that in-place update doesn't require the entire spine of the tree to be immediately reconstructed, unlike a traditional functional tree update. Zippers defer this reconstruction until the tree is explicitly navigated upward. Navigating the zipper upward causes the zipper to *close* the node, rebuilding the spine from the local information stored in the zipper. Figure 4.13 shows the API that zippers provide. The web-world library uses zippers to implement the contextual focus feature of views, and all the navigational operations of views are built on top of the core Zipper API.

4.7 Extending the world with the Foreign Function Interface

The *view-bind* function provides a simple mechanism to connect web-world programs with DOM node events. However, this mechanism alone does not capture arbitrary JavaScript events. For example, on browsers that support the W3C GeoLocation API [41], the browser can notify programs when the physical location of the environment shifts to a different latitude and longitude. The API provides a type for geolocation callbacks,

$$geolocation\_callback : \{\ latitude : double,\ longitude : double,\ \ldots\ \} \to void$$

and two functions *navigator.geolocation.watchPosition* and *navigator.geolocation.clearWatch* to register and clear a callback with the browser:

$$navigator.geolocation.watchPosition : geolocation\_callback \to watchId$$

$$navigator.geolocation.clearWatch : watchId \to void$$

Similarly, external JavaScript sources such as Google Maps provide APIs for embedding external views of the embedded application, as well as access to application-specific events.

Figure 4.13 Zipper API.

Assuming a type *node* defines two functions to inspect its contents, *node-open* and *node-close*:

node-open : *node* → *(listof node)*	Get the children of the node.
node-close : *node (listof node)* → *node*	Reconstruct the node with a new set of children.

then a zipper provides the following navigation:

make-zipper : *node* → *zipper*	Create a zipper focused at the node.
left : *zipper* → *zipper*	Move the focus to the next sibling.
right : *zipper* → *zipper*	Move the focus to the previous sibling.
up : *zipper* → *zipper*	Move the focus to the parent.
down : *zipper* → *zipper*	Move the focus to the first child.
insert-left : *zipper node* → *zipper*	Insert as the previous sibling.
insert-right : *zipper node* → *zipper*	Insert as the next sibling.
replace : *zipper node* → *zipper*	Replace the focused node with the given one.
delete : *zipper* → *zipper*	Remove the focused node.
node-at : *zipper* → *node*	Get the node at the focus.

In the general case, JavaScript APIs provide asynchronous interfaces that signal an event's activation by callback. In these situations, the events are not DOM events, but still should be able to drive World programs. The open nature of the browser environment motivates a foreign function interface (FFI) to bridge web-world programs to these APIs.

4.7.1 API

The FFI provides basic services to bind callbacks from JavaScript into the Racket language, therefore allowing World programs written in Racket to use them. It enables:

1. explicit coersion of JavaScript functions to Racket functions,
2. World extensions to cooperate with arbitrary asynchronous JavaScript APIs, and
3. explicit coersion of values between the hosted (Racket) and hosting (JavaScript) languages.

Figure 4.14 shows a selection of the FFI. The library is intended to be thin, so it does no automatic coersion of values between the hosted (Racket) and hosting (JavaScript) environments. The basic motivation is to provide a low-level layer that library writers use to build higher-level services. The low-level control necessary to effectively bind to JavaScript provides opportunity to break the abstractions of the evaluator's runtime, so these functions are only intended for library writers.

The *js-function→procedure* can lift arbitrary functions[5] from the hosting JavaScript language so they can be called from Racket. In order to provide an extensible hook to add new event types into the World, the API provides a *make-world-event-handler* function. This returns a world handler that can be used as a standard world event handler (e.g. *on-tick*, *on-key*). *make-world-event-handler* takes as inputs two procedures to manage the lifetime of the handler. The first function *up-proc* initializes a callback with the host JavaScript environment, and the second function *down-proc* releases resources when the *big-bang* shuts down. These

[5]The string representation of functions may also be used.

Figure 4.14 Foreign Function Interface for web-world.

js-function→procedure : *js-function* → *procedure*	Lift a JavaScript function.
js-async-function→procedure : *js-function* → *procedure*	Lift a JavaScript asynchronous function.
make-world-event-handler : *up-proc down-proc* → *handler*	Create a new World event handler.
up-proc : *js-function* → *X*	Initialize a general world event handler.
down-proc : *js-function X* → *void*	Shut down a general world event handler.

two procedures both take a special *js-function* which is a JavaScript callback constructed internally by the FFI. The application of this callback schedules a new event to be processed by a running *big-bang*.

The code in Figure 4.15 demonstrates how the FFI can bind the GeoLocation APIs for use in world programs. The functions *start-up-geo* and *shut-down-geo* represent the low-level machinery necessary to register a callback with the underlying JavaScript environment, and the code uses these two functions to create a new world event handler called *on-geo-change*. When a *big-bang* initializes with an *on-geo-change*, the world library synthesizes an appropriate JavaScript callback, initializes the *on-geo-change* handler by calling *start-up-geo*, and then starts the world event loop dispatch. Uses of the callback function by the JavaScript environment introduce new events onto the World event loop.

The *make-world-event-handler* function plays a similar role to the *receiverE* mechanism used in the Flapjax functional-reactive programming language [34] to bridge imperative, void-returning callbacks to functional event-driven frameworks. Both the FFI of this section and Flapjax integrate with external JavaScript services by exposing a value in to the hosting language that sends event values back to a functional event-driven runtime. One small difference from Flapjax is that the World library takes explicit responsibility for the lifetime management of the event handler because World computations can pause or terminate, whereas Flapjax programs do not terminate.

4.7.2 Example: Google Maps

We demonstrate another binding to an external JavaScript API: the Google Maps API [19]. Google Maps provides a sophisticated API for presenting an interactive map, with its own event-handling for drags and mouse clicks. Figure 4.16 shows an example of a web-world program that uses a Google map in conjunction with other web-world elements. The map is not just used for presentation, but also controls the web-world program. A click on the map produces an event that the *click* function reflects onto the world. The remainder of this section describes how to define the functions *make-dom-and-map* and *on-map-click* used in this example.

Unlike the GeoLocation API, Google Maps is not built into to the browser, but instead is dynamically instantiated. This instantiation itself is an asynchronous operation. A use of *js-function->procedure* alone does not have sufficient power to safely bind to asynchronous functions because it does not know to wait until the API's instantiation has completed. For this reason, the FFI function *js-async-function->procedure* presents an explicit mechanism to wait for the pending computation. With it, the bound JavaScript function consumes the normal function arguments passed during function application, as well as two additional arguments that represent success and failure continuations. The JavaScript function takes responsiblity to call the success continuation under normal operation, and if an error occurs, to call the failure continuation. In either case, the Racket evaluation context captures the current continuation and aborts, as discussed in Section 3.4. Effectively, this allows the computation to wait for either of the continuations to be called, therefore exposing asynchronous JavaScript as blocking, synchronous functions.

Figure 4.17 shows the Google Maps API loading code. The code demonstrates two salient features of *js-async-function->procedure*: (1) the ability to handle asynchronous APIs, as used in *initialize-google-maps-api!*, and (2) the ability to return multiple values back to the continuation, as used in *make-dom-and-map*.

Figure 4.15 Binding to the GeoLocation API to World programs with the FFI.

```
;; start-up-geo: js-function → number
;; Initialize the JavaScript GeoLocation API.
(define start-up-geo
  (js-function->procedure "
   function(locationCallback) {
     var watchId = navigator.geolocation.watchPosition(
         function(evt) {
             locationCallback(evt.latitude, evt.longitude);
         });
     return watchId;
   }"))

;; shut-down-geo: js-function number → undefined
;; Disable the JavaScript GeoLocation API.
(define shut-down-geo
  (js-function->procedure
   "function(locationCallback, watchId) {
      navigator.geolocation.clearWatch(watchId); }"))

;; on-geo-change: world-handler
;; Creates a new event handler.
(define on-geo-change
  (make-world-event-handler start-up-geo
                            shut-down-geo))

;; The world is a position pair lat/lng.
(define-struct pos (lat lng))

;; move: world view number number → world
;; Update the known current physical position of the environment.
(define (move w v lat lng)
  (make-pos lat lng))

(big-bang (make-pos 0 0)
          (on-geo-change move))
```

Figure 4.16 Example using Google Maps in a web-world program. This uses FFI definitions in Figure 4.17 and Figure 4.18. The world is a list of two values, representing the latitude and longitude of the last map click.

```
;; Definitions for make-dom-and-map and on-map-click are in Figure 4.17 and Figure 4.18.

;; Creates the dom element and its internal state.
;;
;; THE-DOM is a div whose contents are managed by Google Maps.
;;
;; THE-GMAP is an opaque value that is also managed by Google Maps and
;; can be used to build new world event handlers.
(define-values (THE-DOM THE-GMAP)
  (make-dom-and-map a-latitude a-longitude))

;; on-map-click: world-handler
;; Creates an on-map-click associated to THE-GMAP, ready to be used in
;; a big bang.
(define on-map-click (make-on-map-click THE-GMAP))

;; draw: world view → view
;; Inject the contents of the world into the "template" paragraph.
(define (draw world view)
  (update-view-text (view-focus view "template")
                    (format "~a" world)))

;; click: world view number number → world
;; Respond to clicks on the map by saving the latitude and longitude
;; into the world.
(define (click world view lat lng)
  (list lat lng))

(define THE-INITIAL-VIEW
  (->view
    `(div (p (@ (id "template")) "fill me in")
          (hr)
          ,THE-DOM      ;; ← The dom element managed by Google Maps
          (hr))))

(big-bang (list 'unknown-latitude 'unknown-longitude)
          (initial-view THE-INITIAL-VIEW)
          (to-draw draw)
          (on-map-click click))
```

The *make-dom-and-map* function produces two values: a DOM element that can be injected into the *view* of a web-world program, and an opaque *gmap* value that can be used to register a program to listen for events on the map. It's important to note that the underlying Google Maps implementation takes responsibility for rendering the contents of the DOM element. *web-world* accommodates this shared responsibility through the design of the View API from Section 4.4.

With this support, the fundamental approach used in the GeoLocation example of Figure 4.15 can be adapted to bind world event handlers to events on a Google Map (Figure 4.18). The main difference between Figure 4.18 and Figure 4.15 is the use of closures to account for the JavaScript functions not being closed over the free variables in the enclosing Racket code. It's necessary for the *make-on-map-click* function to produce closures over the *a-gmap* free variable, so that the *setup* and *shutdown* functions can pass the necessary arguments to the raw FFI-bound functions *raw-setup* and *raw-shutdown*.

4.8 Relationship to Clean's I/O library

The approach being taken in *web-world* is closely related to the I/O Library [2] of the functional language Clean. In the Clean I/O model, event handlers consume both the world and an *iostate* argument, and return a tuple of the resulting world and *iostate* GUI state.

$$\textit{event-handler} : \textit{event-information} \ldots \ \textit{world}\ \textit{iostate} \rightarrow (\textit{world} \times \textit{iostate})$$

web-world presents a similar scheme, with the pair of handlers:

$$\textit{event-handler} : \textit{world}\ \textit{view}\ \textit{event-information} \ldots \rightarrow \textit{world}$$
$$\textit{to-draw} : \textit{world}\ \textit{view} \rightarrow \textit{view}$$

There are a few technical differences. In the Clean I/O library, the use of its type system enforces a uniqueness constraint that ensures the output of the functions is dependent on the input. In *web-world*, the nonce value included in the view enables the runtime library to detect a similar kind of dependency. However, *web-world* also permits the output view to have no dependency on the input view, since a program may want to switch from one view to another to indicate a modal change. Another difference, and one that *web-world* shares with World, is that *web-world* uses a concrete type for the view rather than an abstract one; this can make it easier to test functions on views without having to start up an event loop.

From a user perspective, dividing responsibility for generating the new world and GUI state into two functions can make *web-world* easier for beginners to use, since it does not requires knowledge of tuples or structured data, and both functions can be independently tested. On the other hand, in Clean's system, when an event is being handled, the computation for the new *iostate* can use the event information. However, in *web-world*, that event information isn't present in a *to-draw*, and if the view computation does need to know about the reason why the world has changed, then the world updater needs to store it within the new world.

Figure 4.17 Low-level support for Google Maps using the FFI, Part 1. This code installs basic support for Google Maps.

```
;; initialize-google-maps-api!: → void
;; Dynamically loads the Google Maps API.
(define initialize-google-maps-api!
  (js-async-function->procedure "
function(success, fail) {
    window.afterGoogleMapsInitialized = function() {
        delete(window.afterGoogleMapsInitialized);
        success(VOID);
    };
    var script = document.createElement('script');
    script.type = 'text/javascript';
    script.src = 'http://maps.googleapis.com/maps/api/js?key= ...&sensor=false'
        + '&callback=afterGoogleMapsInitialized';
    document.body.appendChild(script);
}"))

;; Initialize Google Maps API support. Wait until the API has been
;; fully instantiated before continuing.
(initialize-google-maps-api!)

;; make-dom-and-map: number number → (values dom-node gmap-object)
;; Dynamically creates both a dom-node and a gmap object. The map centers
;; on the given lat/lng coordinates.
(define make-dom-and-map
  (js-async-function->procedure "
function(success, fail, lat, lng) {
    var myOptions = { center: new google.maps.LatLng(lat, lng) };
    var domElement = document.createElement('div');
    var map = new google.maps.Map(domElement, myOptions);

    // Return two values to the success continuation:
    success(domElement, map);
}"))
```

Figure 4.18 Low-level support for Google Maps using the FFI, Part 2. The function *make-dom-and-map* can construct new world handlers that respond to mouse clicks on a Google Map.

```
;; make-on-map-click: gmap-object → world-handler
;; Creates a new world handler from the second value produced
;; by make-dom-and-map.
(define (make-on-map-click a-gmap)
  (define (setup send-event-callback) (raw-setup a-gmap send-event-callback))
  (define (shutdown maps-listener) (raw-shutdown a-gmap maps-listener))
  (make-world-event-handler setup shutdown))

;; raw-setup: gmap callback → maps-listener-object
(define raw-setup
  (js-function->procedure "
function(map, callback) {
   var mapsListener =
      google.maps.event.addListener(map, 'click', function(event) {
         callback(event.latLng.lat(), event.latLng.lng());
      });
  return mapsListener;
}"))

;; raw-shutdown: gmap maps-listener-object → void
(define raw-shutdown
  (js-function->procedure "
    function(gmap, mapsListener) {
      google.maps.event.removeListener(gmap, mapsListener);}"))
```

Chapter 5

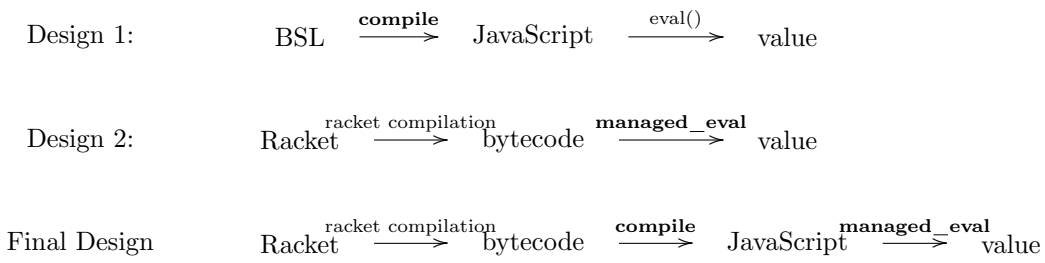
Implementation

The World programming models described in Chapters 3 and 4 need to be executable on web browsers. However, the native capabilities of the language evaluator for browsers—JavaScript—pose difficulties to the running of World programs. An extended evaluator that runs on top of JavaScript must provide additional support. To wit, it must:

- allow programs that use recursion (both iterative and non-iterative) to avoid a browser's stack ceiling,
- allow evaluation to pause and resume to cooperate with the asynchronous JavaScript environment (e.g. to receive events from the browser into World programs), and
- present good error messages and present stack traces in terms of the source program.

It should also evaluate programs efficiently, and although the evaluator's focus is on functional programs, the evaluator should also be able to support imperative features as well. This chapter describes its implementation.

Figure 5.1 Summary of evaluator designs.



The elements in boldface are those that are implemented in this dissertation.

Before the implementation of the current evaluator, two prototype designs were explored with the aim of doing the simplest thing possible to support functional programs. Exploration with the prototypes exposed flaws in the initial approaches. The first prototype could not support certain functional programs soundly because of JavaScript asynchronous issues and stack ceiling collisions, and was limited to a small language subset. The second prototype aimed at a more full-featured language, and could support the functional programs that had defeated the first prototype, but at the cost of a harsh performance hit. The final design, which has been implemented, uses the lessons from the first two prototypes, and in a very real sense is an amalgamation of these prototypes (Figure 5.1).

5.1 Overview of prototype design #1

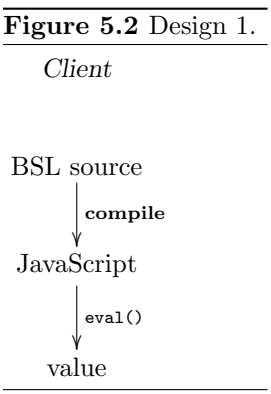


Figure 5.2 Design 1.

The first prototype of the evaluator works on a well-defined subset of Beginner Student Language (BSL); the limited scope of the language made it easier to develop the prototype. BSL is a subset of the Racket language that is tailored toward beginners by disallowing features such as mutation. In the first prototype (Figure 5.2), the compiler, shown in the figure as **compile**, consumes this simplified BSL and produces JavaScript programs that can be directly evaluated. The language is simple enough that the compiler runs on the client side. To make it easier to connect to external JavaScript libraries and for ease of implementation, the first prototype translated BSL into representations that corresponded closely with JavaScript. For example, it does not use a numeric tower or support different numeric types, and instead only uses the native JavaScript floating-point type.

The compiler translates Racket functions to JavaScript functions, and maps function application to JavaScript function application with its standard calling convention. One advantage of this choice is a relatively lightweight cost of evaluation, because the emitted code is in a form that is close to the expectations of the browser's JavaScript engine. Because such translations can be executed efficiently, this translation approach is used in projects like Scheme2JS [26] where performance is a major concern.

However, the implementation exposed several limitations which hampered the evaluator's applicability to World-based functional programs.

- JavaScript provides no native mechanism for suspending execution. Suspension is required to implement synchronous functions such as *big-bang* (Section 3). One role of suspension is to give control back to the browser, so that it can redraw the DOM tree, support animations, and allow interaction with other elements on the web page, all critical features of World programs.

  Suspension is also crucial for properly dealing with asynchronous APIs. This problem instantiates in the form of the implementation of *image-url* (Section 3.3): in the first evaluator, programs using it would occasionally fail, even to the point of exposing browser bugs and crashing!

- By reusing JavaScript's stack for function application, the compiled code can exhaust the JavaScript stack when evaluating recursive definitions, even if the recursion is iterative in nature. Many popular JavaScript evaluators have stack ceilings that are short: earlier versions of Safari were bound to about 500 activation records before it overflows, and IE and Firefox in the low thousands [45].

  Even if JavaScript evaluators were to have tail calls, a related problem still exists with regards to functions that are non-tail recursive.

- The evaluator depends on the JavaScript implementation to produce stack traces during runtime errors, but some JavaScript evaluators don't produce a usable trace. Even when they do provide a trace, that trace is in terms of the compiled program, and not in terms of the original source text.

- The language encompasses only a limited subset of features, and its front-end does not allow reuse of any of the existing infrastructure from Racket, including Racket's rich macro system or any of the existing Racket libraries.

Loitsch [25] provides a solution to the problem of suspending execution in JavaScript by using A-normal form and exception-handling to capture the control context. Not only does this allow the control context to be frozen, but it also provides a solution for tail calls: restoring the control context can omit frames associated to tail calls.

Loitsch's approach takes advantage of the native JavaScript stack by reusing the existing JavaScript function calling convention in the simple case. However, because the activation records use the JavaScript stack, stack overflow continues to be a possibility. Programs that are intrinsically not tail recursive, such as functions that act on trees, can run into the stack ceiling when deep recursion occurs. This makes the approach unusable in a functional context.

5.2 Overview of prototype design #2

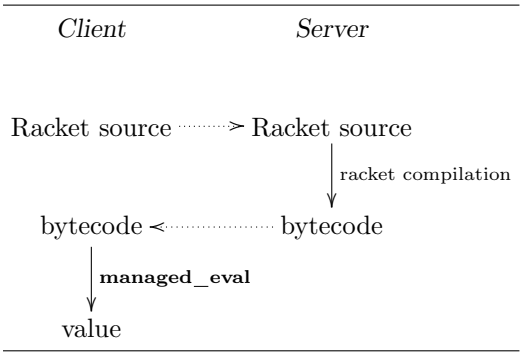


Figure 5.3 Design 2. Dotted lines represent moving code between the client and server.

The second prototype (Figure 5.3) expands the supported language, from Beginner Student Language to full Racket, by implementing a bytecode interpreter for Racket's native bytecode format. This bytecode is produced by the official Racket compiler, but since the Racket compiler is inaccessible on a client's computer, the prototype uses a server to provide a compilation service. The server takes the user's program and compiles it using Racket's compiler, which produces a bytecode stream that can be evaluated using the bytecode interpreter on the browser. The approach allows reuse of an industrial-strength compiler and bytecode design, and also allows the option to use the language-extension features specific to Racket.

The bytecodes generated by the compiler are those of the Racket virtual machine [23]. The reader can think of this as analogous to the bytecodes of the Java Virtual Machine, though in fact the Racket language is richer in many ways and can thus support a host of other programming languages ranging from Java to Python (the Racket project having experimental support for these other languages). Design 2 has been implemented in terms of a bytecode interpreter in JavaScript, indicated as **managed_eval** in Figure 5.3, which relies on the threaded virtual machine technology [16] of most contemporary browser JavaScript implementations to optimize uses of the interpreter.

The inner loop of the interpreter dispatches on the type of the bytecode operations, with a subroutine threading approach [6] to reduce the cost of the dispatch. The interpreter manages the control context as a JavaScript array, which allows the language to evaluate recursive definitions without exhausting the JavaScript stack. This design gives a fairly trivial way to suspend evaluation, since the control context is an explicit value. The ability to suspend and restart evaluation allows *big-bang* to be used as a function that produces values. As a side benefit, the runtime also gets the ability to generate proper stack traces and preserve the source correlation for error messages. To support the kind of stack inspection necessary to generate stack traces, it implements a continuation-marks model [8] to maintain a set of key-value pairs associated to the dynamic calling context; these values are a part of the control context, and are maintained in a way that respects tail calls.

A disadvantage of interpretation is its overhead due to the cost of opcode dispatch. Although modern browsers analyze JavaScript code to improve its performance, the bytecode interpreter doesn't expose the program's control flow in a direct way that can be easily optimized. The second prototype's interpreter is still fast enough to allow basic animations and programs to run, but the overhead makes it run poorly for computationally expensive programs on platforms such as smartphones and tablets.

One factor that penalizes the second prototype's Racket bytecode interpretation strategy is the common evaluation strategy of its hosting language, JavaScript. JavaScript is deployed to client browsers in source-form across the network; as a consequence, there is a high premium on interactivity and perceived latency between the time of downloading a program to executing it. Since there is no server-side compilation process, and because the browser environment is so latency-constrained, browsers often choose interpretation as a major strategy to implement a JavaScript evaluator.

To offset the cost of interpreting JavaScript, a JavaScript evaluator can judiciously use some time and space to compile JavaScript to native code on the client side. This Just-In-Time (JIT) compilation can eliminate interpretive overhead and offer a balanced approach that focuses compilation efforts, not on the entire program, but only on promising regions at run-time.

Mainstream web browsers, such as Mozilla Firefox and Google Chrome, incorporate JIT technology in their JavaScript evaluators. One particular JIT technology, *tracing*, is particularly attractive for dynamic languages like JavaScript. Tracing JITs depend on a few assumptions: that (1) evaluation of loops dominates the runtime of a program, (2) repeated runs follow the same control paths, and (3) the primitive operations

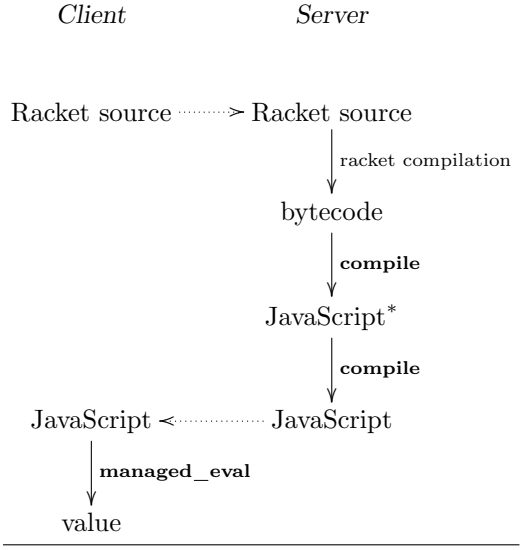
applied during evaluation can be accurately recorded at runtime. In return, when these assumptions hold true, then a tracing JIT can focus its compilation efforts on these loops and expect to eliminating the cost of interpretation and dynamic dispatch.

Typical programs benefit from a tracing JIT. In fact, at a first glance, the first assumption also appears to be true of the prototype evaluator of Figure 5.3, as there is a dispatch loop that serves as the main hotspot of the interpreter. However, the prototype Racket bytecode interpreter is not a typical program. The assumptions that make tracing JITs effective do not easily apply when the program being evaluated is itself an interpreter. Under a basic, naive tracing JIT, the tracer focuses on the interpreter opcode-dispatch loop, which results in traces that are too short to provide a performance benefit. Furthermore, control through the bytecode dispatch loop is unpredicable: a repeated run through a dispatch loop is unlikely to repeat the same operation, which violates the second assumption that tracing JITs depend on. For this reason, the design of Section 5.2 does not gain significant benefit from a tracing JIT.

One way to recover the benefit of a tracing JIT for interpreters is to expose, at a higher level, the control flow of the interpreted program rather than the lower-level one of the interpreter itself. If the program flow is measured as the one represented virtually by the interpreter, rather than by the native program counter, then a tracing JIT may measure more predictable, longer traces. If the tracing JIT provides hooks for the language implementor to provide runtime hints to expose logical loop structure, to virtualize the program counter, then the tracing JIT can effectively construct more effective loop traces that are expressed in terms of loops within the user-level program [39] [7] [43]. Exposing these hints gives the implementor an easy path to integrate a tracing JIT effectively into an interpreter. However, these hinting mechanisms have not yet been incorporated into mainstream JavaScript evaluators.

5.3 Final design

Figure 5.4 Final design.



The final design (Figure 5.4) takes into account the lessons learned from the previous two designs: like the second design, it still reuses the Racket compiler to handle details of macro expansion and linguistic support. Furthermore, it explicitly manages the control context through a **managed_eval** process. Like the first design, it eliminates the cost of interpretation by using a **compile** process to generate JavaScript, exposing the program's natural control flow to the optimizers of typical JavaScript JIT compilers.

The new design matches that of a traditional compiler, with a separate front- and back-end. It treats the Racket bytecode compiler as a front-end into a separate back-end compiler that translates Racket bytecode to JavaScript. It deviates from Design 1 by using traditional compilation techniques, treating the underlying base JavaScript language as a low-level platform with the ability to do labeled jumps. This extended JavaScript, denoted as JavaScript*, is desugared and evaluated in an managing environment that can simulate the necessary features in plain JavaScript.

The back-end compiler's design resembles the register-machine compiler discussed in SICP [1]. It generates low-level basic blocks which can be assembled into JavaScript. The resulting assembled code imposes no interpretive overhead and can be analyzed by the optimizers of modern JavaScript evaluators.

This section explores the final design's two major stages: (1) **compile**, which consumes Racket bytecode and translates to stylized JavaScript, and (2) **managed_eval**, which provides support for this code. The programs being generated by the first **compile** stage use statements like GOTO statements and labels, both which need to be simulated under plain JavaScript. Section 5.3.1 contrasts two basic ways of simulating these

Figure 5.5 Low-level code with GOTO.

```
start:
  if (! (n < N))) {
      goto after-if
  }
  n++
  goto start
after-if:
  ...
```

features (function application, case/switch), and explains why the final design chooses function application. Desugarizing JavaScript* produces the stylized code from **compile**, which expects a rich runtime environment, one that can manage arbitrary control flow, pause evaluation at any time, and permit stack inspection. **managed_eval** provides this support in the form of a virtual machine, described in Section 5.3.2, on which the programs act. To show how the VM enables these features, Section 5.3.3 explores how **compile** and **managed_eval** behave on a representative sample of the bytecodes.

5.3.1 Desugaring JavaScript*: simulating GOTOs and labels

The first phase of **compile** produces programs in JavaScript*, where GOTOs and labels exist to support conditional branching and function call/return. Generic function returns, in particular, introduce indirect jump targets whose target addresses are determined at runtime.[1] Since GOTOs and labels exist in the generated output, **compile** and **managed_eval** must cooperate to simulate them for plain JavaScript.

Figure 5.5 shows a toy example of a program with GOTOs, which will be used to demonstrate the basic approach in desugaring and simulating GOTOs and label addressing. In the absence of a true JavaScript GOTO statement, the second phase of **compile** desugars these features so that they can run on a JavaScript supported by **managed_eval**. Both approaches use a standard basic block analysis to determine the boundaries of each block. Once computed, the basic blocks may be desugared by two approaches:

- By using functions. Each basic block becomes a function named by its label, and each GOTO statement transforms to a function call. Programs that need to reference a label's address use the function value as the reference.

  For example, the `start` basic block of Figure 5.5 translates to lines 3–10 of Figure 5.6.

  The entry point of each transformed block manages the maximum height of the stack. A trampoline (Section 5.3.2), established at the toplevel, ensures that the stack never grows too high as control jumps from one function block to the next.

- By a global case/switch. The content of all the basic blocks are written into a switch, each label is enumerated as a separate case statement, and each GOTO is transformed into a label assignment and a 'continue' to jump to the next basic block.

  For example, the `start` basic block from Figure 5.5 translates to lines 5–12 of Figure 5.7.

Figure 5.8 compares the performance of these techniques; Google Chrome 8.0.552.237 on an Intel i7 1.6ghz system produces the timing data. The case/switch approach has about 50% overhead above a native `for` loop that performs the same work, and functions+trampoline introduces 250% overhead.

The two techniques, however, cannot be applied in every situation: case-switch is applicable only if all the basic blocks are present at compilation time. In contrast, the function-call+trampoline approach can be applied in dynamic linking situations, such as that of interactive evaluation and dynamic module loading.

[1] The translator does perform some limited, ad-hoc optimizations on a small class of direct jumps whose targets are known statically, and more work can be performed to turn direct jumps into native JavaScript structured control flow operators [5] [35], as done in the Emscripten [44] project.

Figure 5.6 GOTO simulation of Figure 5.5 with function calls and a trampoline.

```
var MAX_STEPS = 1000;
var _stepsBeforeTrampoline;
var start = function() {
    if (_stepsBeforeTrampoline-- < 0) { throw start; }
    if (! (n < N)) {
        return afterIf();
    }
    n++;
    return start();
}
var afterIf = function() {
    if (_stepsBeforeTrampoline-- < 0) { throw afterIf; }
    // ...
};
var harness = function(name) {
    _stepsBeforeTrampoline = MAX_STEPS;
    while(true) {
        try {
            name();
            return;
        } catch (e) {
            if (typeof(e) === 'function') {
                name = e;
                _stepsBeforeTrampoline = MAX_STEPS;
            } else { throw e; }
        }
    }
};
harness(start);
```

Figure 5.7 GOTO simulation of Figure 5.5 with `case/switch`.

```
var switched = function() {
    var label = 0;
    while (true) {
        switch(label) {
        case 0:
            if (! (n < N)) {
                label = 1;
                continue;
            }
            n++;
            label = 0;
            continue;
        case 1:
            return;
        }
    }
}
switched();
```

Figure 5.8 Measurements for GOTO simulation of Figure 5.6 and Figure 5.7.

	$N = 1000$	$N = 10000$	$N = 100000$	$N = 1000000$	$N = 10000000$
METHOD	time	...	...	...	...
native	0.03 (0.17)	0.09 (0.29)	0.1 (0.00)	9.9 (0.30)	99.04 (1.37)
case/switch	0.01 (0.10)	0.14 (0.34)	1.41 (0.49)	14.4 (0.89)	142.94 (3.50)
function/100	0.05 (0.22)	0.51 (0.50)	4.99 (1.19)	49.47 (3.12)	474.63 (9.32)
function/10000	0.04 (0.20)	0.36 (0.48)	3.53 (0.52)	34.75 (0.96)	345.1 (1.39)

Time is in milliseconds, followed by the standard deviation. function/100 and function/1000 use trampoline ceilings of 100 and 10000 activation records respectively.

Figure 5.9 The structure of the virtual machine.

$$M = (\text{val}, \text{proc}, \text{argcount}, \text{env}, \text{control}, \text{modmap}, \text{tramp})$$

val : value register
proc : function register
argcount : argument count register
env : environment stack
control : control stack
modmap : module map
tramp : trampoline count register

Because the case/switch technique requires a whole-program transformation that is not easily applicable for domains such as an interactive evaluator on the Web, the second phase in **compile** generates function blocks, and **managed_eval** provides the necessary support for calling them.

5.3.2 Managed evaluation with the virtual machine

The final design requires **managed_eval** to provide environmental support for **compile**-generated programs. **managed_eval** is built on top of a VM that resembles the Racket VM [23] machine. The machine consists of the 7-tuple M, shown in Figure 5.9. The runtime maintains a single machine M and mutates it throughout the course of evaluation. Machine M consists of an explicit representation of the environment (M.env) and control (M.control) stacks in terms of JavaScript arrays, as well as a few registers (M.argcount, M.proc) to handle function calling. The module map register (M.modmap) is specific to the modeling of a Racket-running machine, and holds a mapping from module identifiers to their bindings and exports.

Expressions typically write their value into the value register (M.val), which allows for a fairly straightforward function calling convention; a call typically loads operand values on the stack, initiates a jump to the function's entry point, and expects the return value to be placed into M.val. However, this story is somewhat complicated in order to accommodate multiple return values.

The program counter (PC) of the VM is implicit through the currently running JavaScript function. The entry point of a program running under the VM is a 1-argument JavaScript function that is passed to a trampoline: this function is expected to consume the machine M as an argument and act on the VM by either manipulating M's registers or passing M to other known 1-argument function. The granularity in which the virtual PC can be directly expressed is coarse, but still capturable since a named function can refer to itself.

The trampoline

All evaluation happens in the context of a toplevel trampoline; the trampoline starts by calling a JavaScript function within a `try/catch`; each function constructed by the Racket-to-JavaScript compiler consumes the machine M as its only argument, and the head of the function decrements and checks the machine's M.tramp trampoline-counting register: if its value decrements below zero, the function `throw`s itself as an exception value back to the trampoline. This gives the trampoline two options:

1. The trampoline may immediately reset the counter M.tramp and restart the computation.
2. The trampoline may schedule a restart of the currently running computation using a `setTimeout`. This enables cooperative multitasking by yielding control back to the browser.

In either case, the `throw` allows the runtime to discard stack frames on the native JavaScript stack. The toplevel exception handler monitors other exception types to implement operations like continuation capture or to translate low-level JavaScript errors into Racket errors.

The second case of pausing evaluation with `setTimeout` allows the VM to yield control to the browser, even in the middle of a long-running computation. However, cooperative multitasking permits certain control flow sequences that can be disastrous to the VM if not anticipated. Entry into the trampoline is guarded by an mutually exclusive lock to prevent inadvertent re-entrancy into the VM.

Locking

It may seem absurd to talk about mutual exclusion in the context of JavaScript, which is a single-threaded language. However, although JavaScript is single-threaded, it provides several mechanisms to perform cooperative multitasking, opening the door to threading issues that one might not immediately anticipate. Long running programs in JavaScript (such as the trampoline) will use `setTimeout` every so often to provide a nice user experience: the timeout gives the browser time to perform tasks such as page updates. To view it more defensively: periodically relinquishing control to the browser allows a program to dodge a browser's watchdog, which pre-emptively terminates JavaScript computations that appear to use too much computation.

Figure 5.10 demonstrates a toy program whose behavior depends on the scheduler of `setTimeout`. The program directs functions to subtract from some shared value and later restore the value back. Each helper function waits its turn by using `setTimeout` for some random interval. At the end of this program's execution, the shared value is intended to sum to its original value `50`, but running the program several times can produce radically different results, such as `31`, `36`, and other nonsensical values. The use of `setTimeout` allows multiple "threads" to contend for some shared state, and each "thread" of execution can inadvertently interfere with another because of the unpredictability of the `setTimeout` scheduler.

The `setTimeout` function is one way to create contention, but JavaScript is rife with asynchronicity. Although the example in Figure 5.10 is artificial, the fundamental problem exists, and users have observed mutual-exclusion issues in earlier versions of the evaluator. The use of the trampoline to cooperate with the browser, as well as the adaptation of event-driven programming for World programming, both give opportunities for multiple "threads" of execution to make changes to the VM and clash with inopportune interleavings.

In order to resolve these issues, a language often provides mechanisms for delimiting regions of mutual exclusion. JavaScript does not provide built in mechanisms, but since JavaScript is a single-threaded language, the implementation of exclusion can be much simpler than in a typical threaded environment. Figure 5.11 shows **managed_eval**'s implementation of mutual exclusive locks; when applied to the program in Figure 5.12, the program produces consistent results. Within **managed_eval**, the trampoline uses the `ExclusiveLock` class to ensure multiple computations do not trample over each other, even through they run on the same virtual machine.

Figure 5.10 A toy program that demonstrates a race-condition in JavaScript.

```
1  // generate a random integer between [0, n)
2  var randInt = function(n) {
3      return Math.floor(Math.random() * n);
4  };
5
6
7  // schedule a thunk f to be called at some point.
8  var schedule = function(f) {
9      setTimeout(f, randInt(100));
10 };
11
12 var sharedValue = { n : 50 } ;
13
14 // decrement a value in sharedValue, and increment it again.
15 // intended to have no effect on the final value of sharedValue.n,
16 // but the use of schedule() allows certain unexpected control flow
17 // interleavings to occur:
18 var doWork = function() {
19     var amount = Math.floor(randInt(sharedValue.n));
20     sharedValue.n = sharedValue.n - amount;
21     schedule(function() {
22                  var newValue = sharedValue.n + amount;
23                  schedule(function() { sharedValue.n = newValue; });
24              });
25 };
26
27 var afterLoad = function() {
28     var i = 0;
29     for (i = 0; i < 10; i++) {
30         schedule(doWork);
31     }
32
33     // after waiting for the computation to complete...
34     setTimeout(function() { alert(sharedValue.n); },
35                2000);
36 };
```

Figure 5.11 An implemention of mutual exclusive regions in JavaScript.

```
var ExclusiveLock = function() {
    this.locked = false;  // boolean
    this.waiters = [];
};

ExclusiveLock.prototype.acquire = function(onAcquire) {
   var that = this;
   var alreadyReleased = false;
   if (this.locked === false) {
      this.locked = true;
      onAcquire.call(
           that,
           function() {             // releaseLock
               var waiter;
               if (alreadyReleased) {
                  throw new Error("Internal error: already released");
               }
               if (that.locked === false) {
                   throw new Error("Internal error: already unlocked");
               }
               that.locked = false;
               alreadyReleased = true;
               if (that.waiters.length > 0) {
                   waiter = that.waiters.shift();
                   setTimeout(function() { that.acquire(waiter.onAcquire); },
                              0);
               }
           });
    } else { this.waiters.push({ onAcquire: onAcquire } ); }
};
```

Figure 5.12 A revised version of `doWork` from Figure 5.10 that enforces mutual exclusion.

```
var doWork = function() {
    sharedValue.l = sharedValue.l || new ExclusiveLock();
    sharedValue.l.acquire(
        function(release) {
           var amount = Math.floor(randInt(sharedValue.n));
           sharedValue.n = sharedValue.n - amount;
           schedule(function() {
                        var newValue = sharedValue.n + amount;
                        schedule(function() {
                                    sharedValue.n = newValue;
                                    release();
                                 }); }); }); };
```

Figure 5.13 The Racket bytecode format

⟨*Expression*⟩ ::= Top of ⟨*Prefix*⟩ × ⟨*Expression*⟩
  | Module of ⟨*symbol*⟩ × ⟨*Module-Path*⟩ × ⟨*Prefix*⟩ × ⟨*Module-Path*⟩* × ⟨*Expression*⟩
  | Require of ⟨*Module-Path*⟩
  | Constant of ⟨*Value*⟩
  | ToplevelRef of ⟨*depth*⟩ × ⟨*index*⟩
  | LocalRef of ⟨*depth*⟩
  | ToplevelSet of ⟨*depth*⟩ × ⟨*Expression*⟩
  | Branch of ⟨*Expression*⟩ × ⟨*Expression*⟩ × ⟨*Expression*⟩
  | Lam of ⟨*symbol*⟩ × ⟨*id-list*⟩ × ⟨*closure-map*⟩ × ⟨*Expression*⟩
  | CaseLam of ⟨*Lam*⟩*
  | EmptyClosureReference of ⟨*id-list*⟩ × ⟨*symbol*⟩
  | Seq of ⟨*Expression*⟩+
  | Splice of ⟨*Expression*⟩+
  | Begin0 of ⟨*Expression*⟩+
  | App of ⟨*Expression*⟩+
  | Let1 of ⟨*Expression*⟩ × ⟨*Expression*⟩
  | LetVoid of ⟨*natural*⟩ × ⟨*Expression*⟩
  | LetRec of ⟨*Lam*⟩* × ⟨*Expression*⟩
  | InstallValue of ⟨*depth*⟩ × ⟨*Expression*⟩
  | BoxEnv of ⟨*depth*⟩ × ⟨*Expression*⟩
  | WithContMark of ⟨*Expression*⟩ × ⟨*Expression*⟩ × ⟨*Expression*⟩
  | ApplyValues of ⟨*Expression*⟩ × ⟨*Expression*⟩
  | DefValues of ⟨*ToplevelRef*⟩* × ⟨*Expression*⟩
  | PrimitiveKernelValue of ⟨*symbol*⟩
  | VariableReference of ⟨*ToplevelRef*⟩

⟨*Prefix*⟩ ::= ⟨*PrefixElement*⟩*

⟨*PrefixElement*⟩ ::= ⟨*symbol*⟩
  | GlobalBucket of ⟨*symbol*⟩
  | ModuleVariable of ⟨*symbol*⟩ × ⟨*symbol*⟩

⟨*depth*⟩ :== ⟨*natural*⟩

⟨*index*⟩ :== ⟨*natural*⟩

⟨*id-list*⟩ ::= ⟨*symbol*⟩*

⟨*closure-map*⟩ ::= ⟨*natural*⟩*

⟨*Module-Path*⟩ ::= ⟨*symbol*⟩

5.3.3 Bytecode translation

The bytecode format that **compile** translates generalizes the structures provided by Racket's *compiler/zo-parse* bytecode-parsing library. The extra level of abstraction allows the compiler to run across multiple versions of Racket in spite of internal changes to Racket's bytecode. Figure 5.13 shows the basic structure. Despite its bytecode origins, it forms a tree rather than a sequence. The first phase of **compile** applies a standard case-analysis on its variants, linearizes the tree, and constructs a sequence of JavaScript* instructions, which then desugars to function blocks as described in Section 5.3.1.

The functions that implement **compile** and its helpers consume four arguments (*exp*, *cenv*, *target*, and *linkage*):

1. The *exp* expression argument represents the particular expression to be compiled.

2. The *cenv* compile-time environment argument maintains information about the contents of the environment stack that can be determined statically.

3. The *target* argument tells **compile** to store the result of an expression's evaluation into a particular location. Usually, this is the value register ('val) of the VM. In some cases, **compile** will instruct an expression's evaluation to install the result directly into the environment stack, especially when compiling local-binding expressions such as (**let** ([x ...]) ...) or the operands of a function application.

4. The *linkage* argument describes the control context in which the expression evaluates. Unlike the linkage in the SICP compiler, the linkage is not symbolic, but a structured value that defines (1) how control flow is expected to follow after evaluating the expression, and (2) how many values that context expects. There are three types of linkage:

   (a) *ReturnLinkage*. Control flow should *return* by popping the control stack and following its stored address. The *ReturnLinkage* also holds a flag if the context is in tail position or not, which determines if the the environment should be cleared on jump.

   (b) *NextLinkage*. Control flow should drop down immediately to the following instruction.

   (c) *LabelLinkage*. Control flow should jump to a labeled position in the instruction sequence.

   Each of these linkage types maintains a representation of how many values the context expects to receive. The linkage describes a portion of the continuation, and its value determines how certain expressions are compiled.

The overall structure of the compiler is similar to that of the register machine compiler described in SICP [1]. However, **compile** is generalized to support the stack-oriented nature of the Racket VM and its use of multiple values. Another significant technical difference is that *ReturnLinkage* does not unconditionally imply a tail-calling context, though it still allows label addresses to be reified on the control stack. This feature is used to support the continuation marks feature, discussed in Section 5.3.5.

Because the implementation of **compile** is fairly straightforward but verbose, we describe only a sampling of the compilation of three representative bytecodes: conditional branching (*Branch*), continuation marks (*WithContMark*), and function application (*App*). The following three subsections demonstrate the basic approach, their source code in Appendix A.[2]

5.3.4 Compiling branches

The *Branch* bytecode represents conditional branches, consisting of a test expression, a consequent if the test is true, and an alternative if the test is false (Figure A.1).

It may appear utterly baroque to generate significant code to handle *Branch*. JavaScript has a native `if` statement that, on a first glance, can entirely manage the control flow necessary for *Branch*. However, the

[2]For the full source code of the translation, see `https://github.com/dyoo/whalesong/blob/master/compiler/compiler.rkt`.

Figure 5.14 Compiling to low-level JavaScript.

```
(if <predicate>
    <true branch>
    <false branch>)
...
```

(a) Racket

```
  <assembly of predicate>
  if (! MACHINE.valueRegister)
      goto falseBranch4;
trueBranch5:
  <assembly of true branch>
  goto afterIf3;
falseBranch4:
  <assembly of false branch>
afterIf3:
```

(b) JavaScript*

```
var entry = function() {
  <assembly of predicate>
  if (! MACHINE.valueRegister)
      return falseBranch4();
  <assembly of true branch>
  return afterIf3();
};

var falseBranch4 = function() {
  <assembly of false branch>
  return afterIf3();
};

var afterIf3 = function() { ... };
```

(c) JavaScript

evaluation of any of the subexpressions within *Branch* may need to reliquish control back to the browser. In such situations, the representation of control must be reified. In the most general case, the use of JavaScript's `if` makes a portion of the control state implicit and uncapturable by **managed_eval**.

To generate a linear instruction sequence for *Branch*, the compiler first generates the predicate's code. In doing so, the compiler takes special care to generate code for the predicate that guarantees that it produces a single value. If the compiler can't statically guarantee that the predicate will return a single result, it adds additional runtime checks into the compiled code to raise an error under multiple-value producing scenarios. The code for the predicate writes the expression's value into the value 'val register, and a conditional jump follows, along with the consequent and alternative instruction sequences. The compiler takes special care to preserve the correct linkage to enable tail-call optimization. The code generated by a *Branch* looks roughly like the code in Figure 5.14b, containing both *make-TestAndJump* and *Goto* instructions as well as jump-targeted labels.

5.3.5 Compiling continuation marks

The Racket bytecode format provides a primitive, *WithContMark*, which enables a versatile stack inspection mechanism that is used for features such as error reporting. *WithContMark* allows a program to mark off a region of program execution, the mark lasting for the dynamic extent of the region's execution (Figure A.2).

In the course of evaluation, a program may place a key/value pair—a continuation mark—on the top element of the control stack. One example of such an annotation may be the position of the currently-running expression, such as:

(*with-continuation-mark* 'current-position "line 42, column 16"
  <*body-expr*>)

Other primitive operations within <*body-expr*> can then inspect the control stack in a limited, controlled manner to extract these annotations at runtime.

For example, programs running on the VM can apply *WithContMark* to provide good stack traces whenever an operation needs to raise an error. If all function applications are wrapped with a *WithContMark*

operation, then this mechanism allows programs to generate a precise stack trace that is expressed in terms of the user program rather than the native JavaScript stack.

Mechanically, *WithContMark* evaluates a key/value pair, binds that pair in either the current control frame or a fresh one, and then evaluates its body in the context of that key/value binding. As with the translations of many of the other bytecodes, there is a major choice when compiling *WithContMark* with regards to the *linkage*, to determine whether or not a fresh control frame element needs to be synthesized.

If the expression being evaluated is in a context that returns to a caller (*ReturnLinkage*) then the compiler generates code to (1) evaluate the key and value, (2) store them within the existing control frame, and (3) evaluate the body in that same context. This allows for annotations on the stack that still preserve the space bounds of tail calls. That is, in a function definition like:

(**define** (*f* ...)
  (*with-continuation-mark* 'a-key 3
    (**begin** ... (*with-continuation-mark* 'a-key 4 <*body-expr*>))))

the inner uses of *WithContMark* lies in tail position, and therefore the inner use of *WithContMark* overwrites the key/value binding of the outer use of *WithContMark*.

Otherwise, if *WithContMark* is in a non-returning context, its compiled code must ensure that the annotation does not accidently overwrite one that may live on the top of the control stack. It adds code to synthesize a control frame so that continuation marks can stack on top of each other. The compiled code for the body uses a *ReturnLinkage* so that, once the body of the expression has finished, the control flow can be directed to pop this auxiliary control frame and continue the rest of the computation.

5.3.6 Compiling function application

Function application forms the heart of the compiler. Figure A.3, A.4, and A.5 show a simplified version of the compilation for *App*.[3]

Function application involves almost all of the VM's components. It involves two phases. The first phase (Figure A.3) gathers values for the function and its operands, and then:

1. increases the size of the environment M.env to hold operand values,
2. invokes the code to generate the operand values and store them into M.evn, and
3. stores the function to be applied in register M.proc, and the number of arguments in register M.argcount.

The function and each of the operands are evaluated in left-to-right order. The generated code for each operand uses the *NextLinkage* structure, a non-tail context, so that if the evaluation of an operand involve a nested application, then a control frame will be synthesized and installed into the control stack (M.control) to restore control flow after the operand's evaluation.

Once all the values of the function and operands are computed, the second phase (Figure A.4) transfers control to the the function held in M.proc:

1. For tail calls, clear the stack space of the current call.
2. For non-returning calls, if there is more work to be done once the function application completes, push a frame into the control stack M.control to handle the result of the call.

   Also, establish separate return points on the control stack to resume control once the application completes.
3. Jump into the entry point held in M.proc.

When a function is being applied in a returning context, the generated code takes care not to add new frames to the control stack, and if it is a tail call, to clear out and reuse the environment stack in order to preserve space bounds. Otherwise, it passes control to the callee.

[3]The true implementation also accounts for the open-coding of primitives, the application of statically-known lambdas, and primitive application.

Function calls, tail calls, and multiple return values

After a function call proceeds, the callee takes responsibility for the rest of evaluation, and follows a particular protocol for returning values to the caller. In the single-value case, the callee can send a single value back to the caller by:

1. writing the single result of a function's call to M.val, and
2. transferring control to the continuation by popping a frame from M.control and jumping to its address.

This story becomes somewhat more complicated in an environment that supports Racket, because multiple values can be returned from any function. In the context of functional programming, libraries often take advantage of this feature to simulate mutation on a data structure. For example, Figure 5.15 shows a stack that provides a functional *stack-pop* by returning two values: the topmost element, and the rest of the stack. The first value is the primary return value, while the second value represents the effect to the data structure.

Although the use of multiple values occurs relatively infrequently, it places a runtime responsibility on every returning context. Since multiple values can be returned to any context, even to one that doesn't expect multiple values, a context that receives the wrong number of values must raise a predictable runtime error. The pervasive effects of multiple values demand a careful approach to adapting the function value-returning protocol to them.

We explored three possibilities and constrasted their performance:

- *Structured*

  A common method to support multiple-value return is to use a distinguished structure to represent the act of sending multiple values back to a context. Each context checks whether or not it is appropriate that it receives multiple values.

  One disadvantage of the structural approach is that each context needs to make an explicit check to ensure contents receive the proper number of values. Since adding explicit checks for multiple values to every context imposes a slight overhead, it would be preferable to find alternatives that don't penalize the common case of single value return.

- *Two continuations*

  Ashley and Dybvig [4] implement multiple values without explicit checks. Their idea hinges on code-address pointer arithmetic: a compiler can inject auxiliary instructions to deal with multiple values at a fixed offset behind single-value-handling code. A function that returns a single value takes the address stored in the top element of M.control and jumps. Functions that return multiple values, on the other hand, jump to the fixed offset before the address in the frame. For contexts that expect multiple values, a compiler can inject a block of code at the fixed offset to handle those values, and for contexts that don't expect multiple values, a compiler can inject error-generating code.

  Although JavaScript does not have native support for address arithmetic, it's possible to encode the idea in spirit. A direct way to support this technique is to push a function that corresponds with the multiple-value context alongside the normal return address. Together, this pair allows single and multiple-value return points, but at the cost of doubling the number of stack pushes and pops needed for function application.

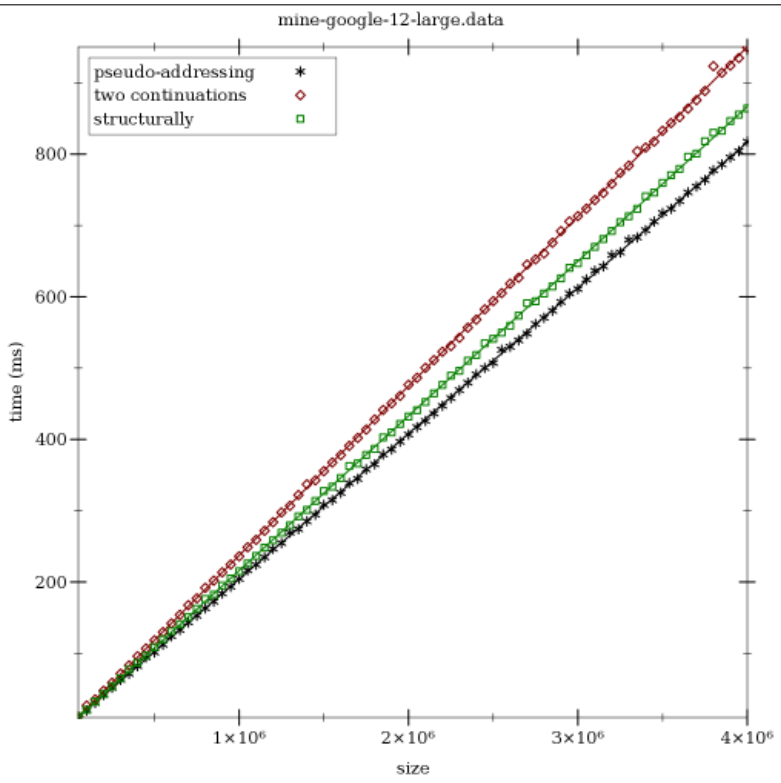
- *Pseudo-addressing*

  JavaScript functions are objects: a limited kind of address arithmetic can be simulated to capture the essense of Ashley and Dybvig's approach without pairs of continuations on the control stack. Because JavaScript objects are mutable and can hold object attributes, a function that corresponds to a return point can be annotated with an attribute to another function. In this way, a limited kind of pointer arithmetic can be supported. This pseudo-addressing technique captures the essential features of Ashley and Dybvig's pointer arithmetic technique without increasing the traffic on the stack.

Figure 5.15 A use of multiple values in the Racket language. Subsequent figures show JavaScript translations of this code, using explicit structures (Figure A.6), multiple continuations (Figure A.7), and pseudo-addressing (Figure A.8).

```
;; stack-push: stack X → stack
;; Push an element onto the stack.
(define (stack-push a-stack elt)
  (cons elt a-stack))

;; stack-pop: stack → (values X stack)
;; Pop an element from the stack, and also return the
;; rest of the stack.
(define (stack-pop a-stack)
  (values (car a-stack)
          (cdr a-stack)))

;; stack-empty?: stack → boolean
;; Produces true when the stack is empty.
(define (stack-empty? a-stack)
  (empty? a-stack))

(define empty-stack '())
(define LIMIT 4000000)

(define (test)
  (define the-stack
    (let loop1 ([the-stack empty-stack]
                [i 0])
      (cond
        [(< i LIMIT)
         (loop1 (stack-push the-stack i)
                (add1 i))]
        [else
         the-stack])))

  (let loop2 ([the-stack the-stack])
    (cond
      [(stack-empty? the-stack)
       (void)]
      [else
       (let-values ([(val the-stack)
                     (stack-pop the-stack)])
         (loop2 the-stack))])))

(test)
```

Figure 5.16 Comparing the cost of multiple value return techniques. Measured on Google Chrome 12 on an 3200Mhz AMD Phenom II X4 995.



Figures A.6, A.7, and A.8 demonstrate these three techniques applied to the code in Figure 5.15.

Figure 5.16 compares the relative performance of each technique on the three translations of Figure 5.15 under Google Chrome. The pseudo-addressing technique demonstrates a lower constant overhead compared to the other two, and other browsers present similar results. Therefore, the evaluator uses pseudo-addressing: in order to enable limited code-pointer arithmetic between blocks, **compile** annotates pairs of return-address labels. When the compiler assembles a JavaScript function named by its label, it also emits an assignment to the `multipleValueReturn` attribute to its predecessor block, effectively tying pairs of these functions together. Later at runtime, given a reference to a function, the evaluator can compute a reference to the previous block by looking at the `multipleValueReturn` attribute.

With the code generated in this way, here is the amended protocol for returning values to the caller.

- In the single-value case, the callee will send a single value back to the caller by:
  1. writing the single result of a function's call to M.val, and
  2. transferring control to the continuation by popping a frame from M.control and jumping to its address.
- In the multiple-value case, a callee will send multiple values (either 0 or more than 1 values) back to its caller by:
  1. writing the first result of a function's call (if any) to M.val,
  2. storing the rest of the results into M.env,
  3. writing a count of the results into M.argcount, and
  4. transferring control to the continuation by popping a frame from M.control and jumping to its `multipleValueReturn` receiver.

5.3.7 Wrap up

The final design's language encompasses the entirety of the Racket bytecode. This makes it possible to fully reuse the Racket compiler infrastructure—its macros and module system—to support different languages with little additional effort, from languages focused on teaching, such as Beginner Student Language, to the professional-focused language, which supports imperative operations and mutation.[4]

All program errors are managed by the language, so that no low-level JavaScript errors are directly exposed to the user. The support provided by **managed_eval**'s maintenance of continuation marks allows program errors to be presented with source locations and stack traces relative to the user's original program.

Figure 5.17 compares the performance of the final design's evaluator on the standard Gabriel LISP benchmark suite [15] relative to native JIT-ed Racket. Initial results show that the performance is within a factor of 50-100X of Racket. This result improves upon the performance of the second prototype's interpreter, which exhibited performance within a factor of 1000-10000X of Racket.

Although these benchmarks show a performance boost over the prototype, there is plenty of room for improvement. A major factor is the overhead of the managed function call. The calling convention allows **managed_eval** to capture the currently-running computation (as well as continuation marks), but not every computation requires that flexibility. In particular, if the compiler can statically deduce that the body of a function avoids continuation capture and its evaluation uses a bounded stack, then its translation may reuse the techniques of the first prototype. It is future work to add this analysis to **compile**, so that it can eliminate, for simple functions, the overhead of **managed_eval**'s function-call calling convention.

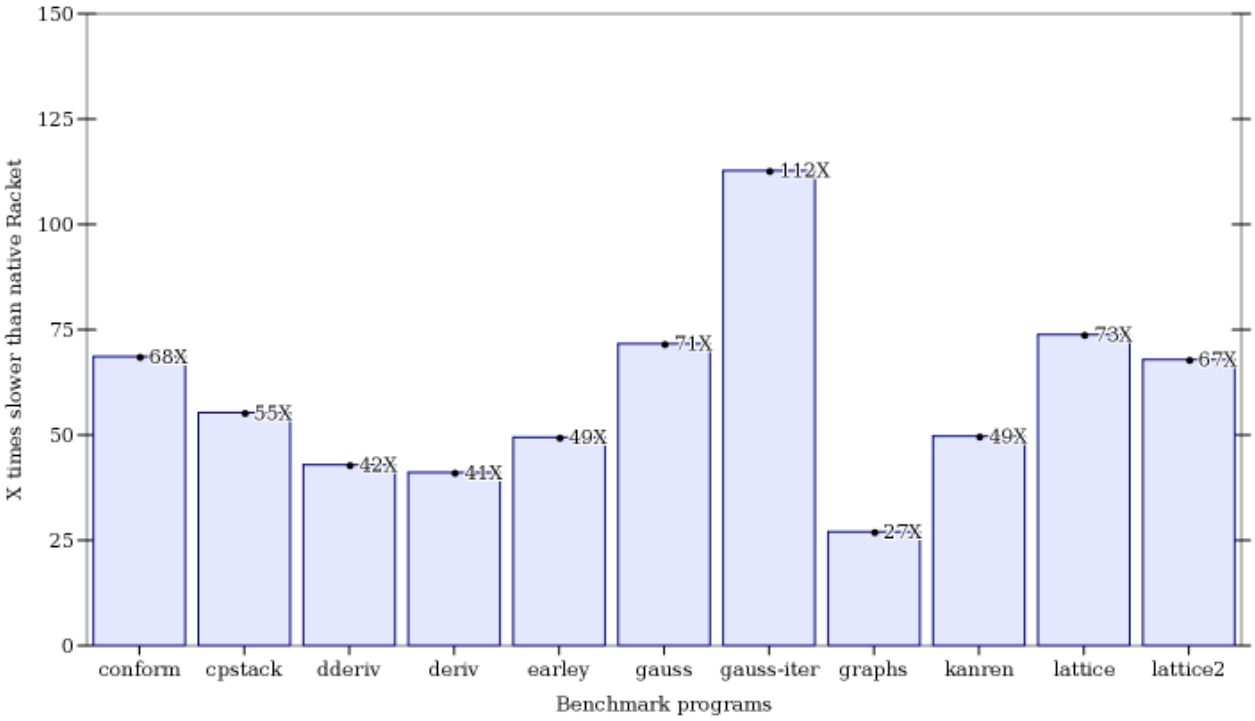
The immaturity of some of the primitive function definitions in the implementation can account for some of the variation in the benchmarks; low-level primitives, such as vector indexing (*vector-set!*) and numeric comparisons ($<$, $=$, ...), have not yet been inlined into the emitted compiled JavaScript. Instead, these

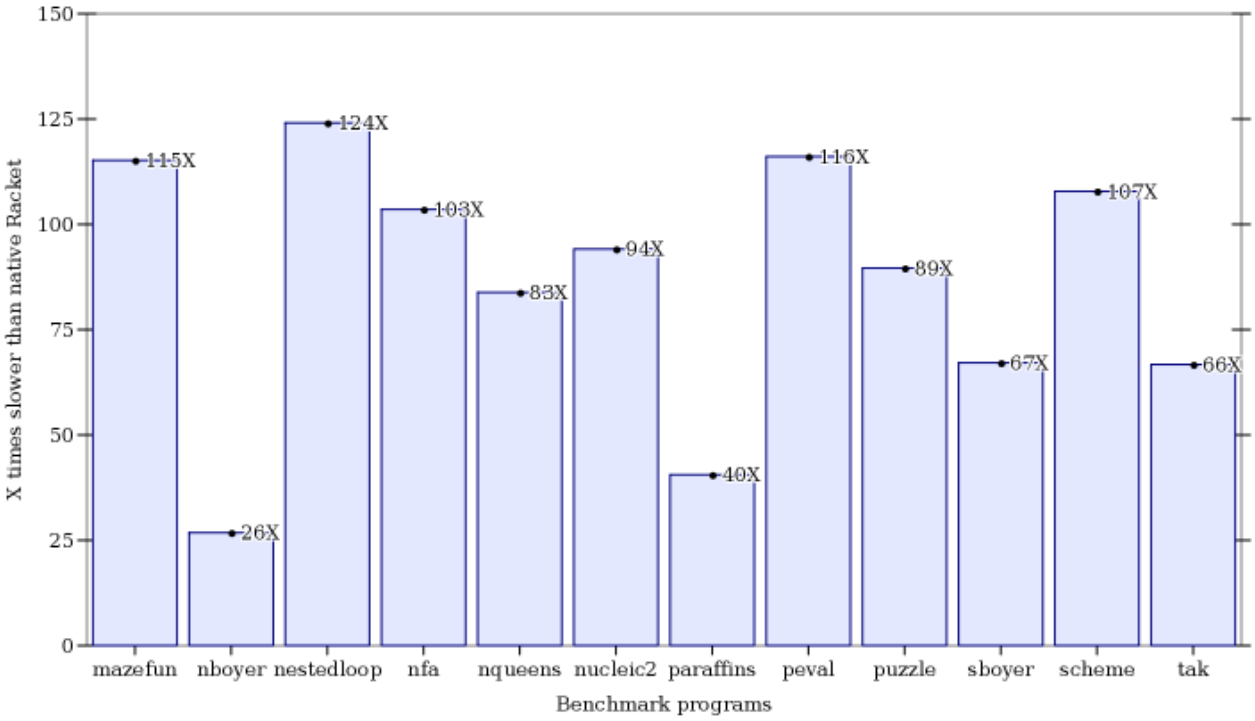
[4]The compiler can even support esoteric languages, which demonstrates the flexibility of the language infrastructure. One such example resides in the standard distribution: `github.com/dyoo/whalesong/tree/master/bf`

primitives currently use a similar function calling convention to the one that's used in general function application.

Another contributing factor to the performance overhead is the numeric tower. The implementation of the numeric tower repeats the work of checking overflow, but mainstream JavaScript evaluators do a similar check to promote numbers from exact integer representations to floating-point. Our implementation of the numeric tower performs a generic type dispatch that has not yet been fully-optimized to improve the fast-path performance for integer and floating-point arithmetic.

Figure 5.17 Performance of the final design's evaluator (relative to natively JITed Racket)





Chapter 6

Deployed Tools: WeScheme and Moby

The evaluator from Section 5 provides a basic environment for running functional, event-driven programs on the Web. This evaluator can be applied to many domains, and a particularly compelling application is WeScheme, a Web-based programming environment for the Scheme [38] and Racket [13] programming languages. A secondary application of the evaluator is Moby, a toolchain for producing Android smartphone packages.

6.1 WeScheme

The motivation behind WeScheme is to provide a simple programming environment that supports algebra-centric curricula such as Bootstrap [37]. WeScheme provides a standalone programming environment, including a syntax-highlighting program editor, an interactive tool to run programs on-the-fly, and a hub for sharing programs. Beneath the surface, WeScheme takes advantage of this dissertation's evaluator and runtime to allow programs to be written in a sequential, synchronous style, a model that is particularly well-suited to beginners for its simplicity.

Figure 6.1 shows a screen-shot of WeScheme running inside the Google Chrome browser. (The environment looks essentially identical in other browsers.) The interface, which borrows heavily from DrRacket [12] (formerly DrScheme), is intentionally simple.

The toolbar at the top has the following commands:

- The Run button loads the program's definitions for use in the REPL.
- The Stop button interrupts the running program.
- The Save button saves the program onto the Cloud.
- The Share button allows the user to freeze the current program and produce a stable URL that refers to it. Accessing the URL presents an interface to look at the source code, if allowed by the owner, and to run the program outside the editing environment.

Below the toolbar is the *Definitions* pane which contains a rich-text program editor (currently a fork of CodeMirror). The editor include a syntax highlighter with color-coding, parenthesis-matching, and context-sensitive indentation, all of which help users with the editing process.

The lower pane contains the REPL. A REPL presents a calculator-like interface to a program. A REPL allows a programmer to explore a program's definitions directly without having to create external binaries. Instead, when the user enters an expression at the prompt, the REPL evaluates it, prints its value, and presents a fresh prompt. The use of a REPL allows for lightweight exploration of programs, and its interface can help cement the relationship between algebra and computation.

Figure 6.1 WeScheme.

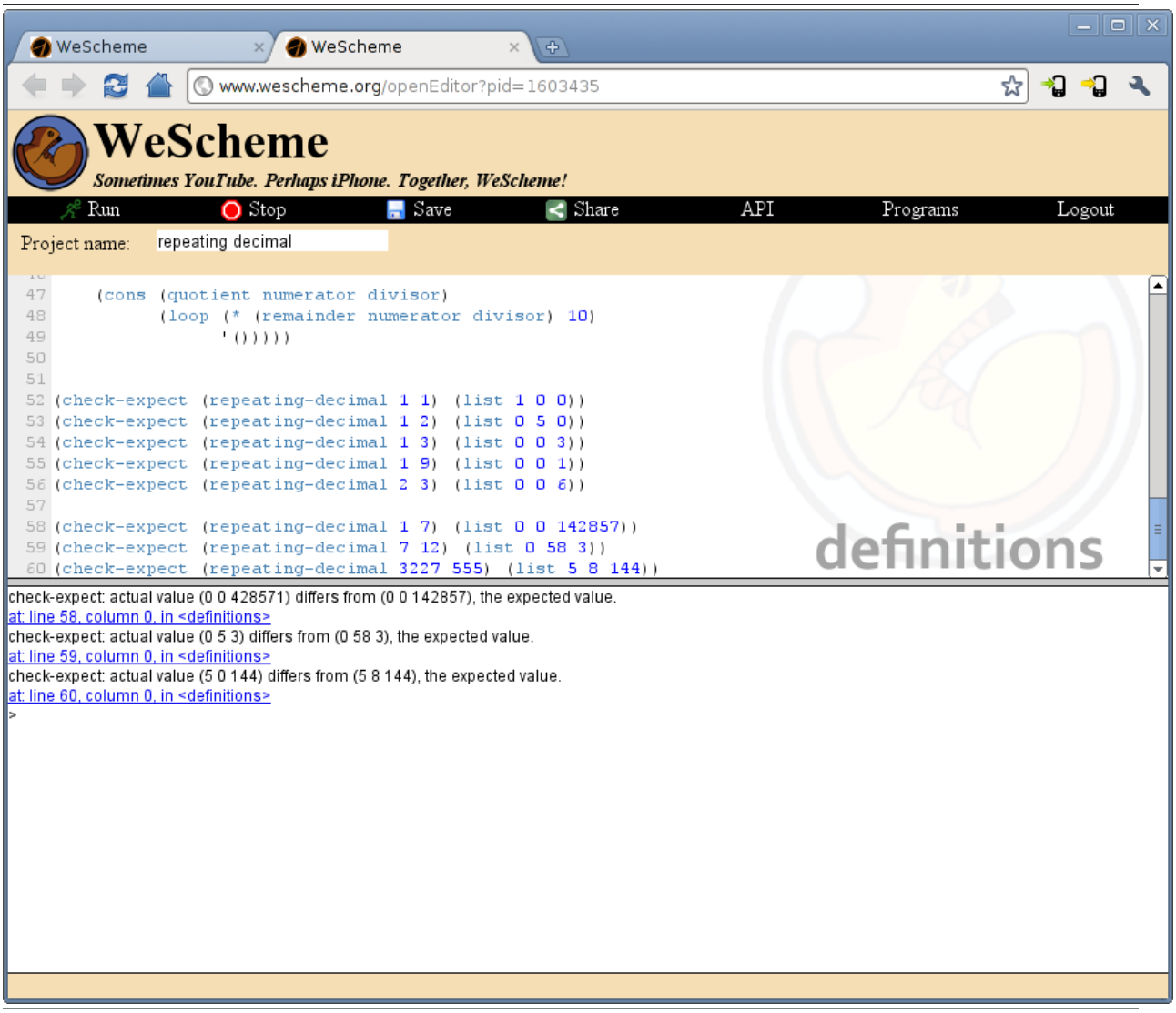


Figure 6.2 Examples in the REPL.

WeScheme's REPL makes heavy use of the browser's display and interaction technology. The screenshot in Figure 6.2 shows four illustrative REPL interactions. In the first, the user is examining a fractional value; in the second, a repeated decimal. In both cases, WeScheme presents them using representations based on those of math textbooks. In the third example, the user evaluates an expression whose value is a list of images, which are all displayed in-line. In all three cases, WeScheme exploits the browser's display technology—JavaScript, canvases, and style-sheets—to present its output. The fourth example shows an error. Error messages are presented as hyperlinks, and clicking highlights the relevant region of code.

Users can browse through their list of programs, and edit, share, and delete them. Figure 6.3 shows such a listing. This list, which takes the place of a typical filesystem, resides on a cloud server, so the user sees the same list no matter where they log in. In the entry for "Baduk", the sharing icon is grey because that program has not yet been shared. When a user chooses to share a program, WeScheme shows the dialog in Figure 6.4. This lets the author choose whether or not to divulge the program source. Once shared, WeScheme generates a stable link; hovering over the sharing link (now green) in the console, as shown in Figure 6.5, provides this link, and also the ability to upload it to various social networking sites to make it easy to share programs with friends. Users who visit the shared URL will see the window in Figure 6.6, where they can run the program and, if allowed, read the source code.

6.1.1 Implementation

WeScheme uses a compiler running on a Cloud server. When the user executes a program, the program is sent to the remote compiler, which converts the source into a compiled output. Likewise, every expression a user types at the REPL is sent to the server for compilation. An evaluator on the client side browser consumes the compiled program and executes it, displaying output on the browser. The use of a compiler on the remote server and evaluator on the local browser adds a network dependency, but in our context of a web-accessible programming environment, this is not onerous. It allows the use of WeScheme without installing a complicated toolchain on the local machine, while still providing the benefits of interactive evaluation, as

Figure 6.3 Program List.

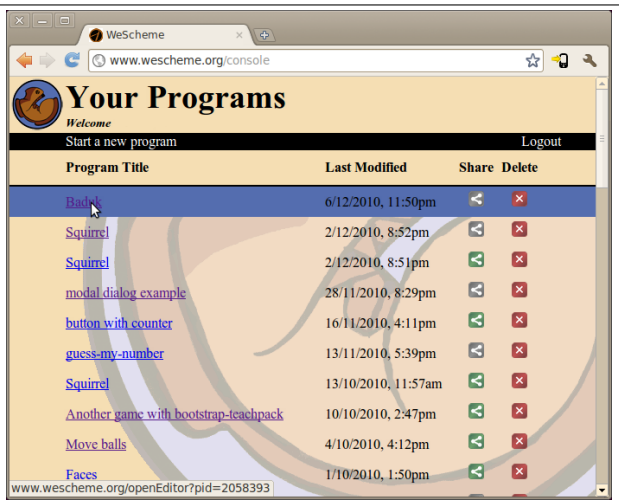


Figure 6.4 Sharing Dialog.

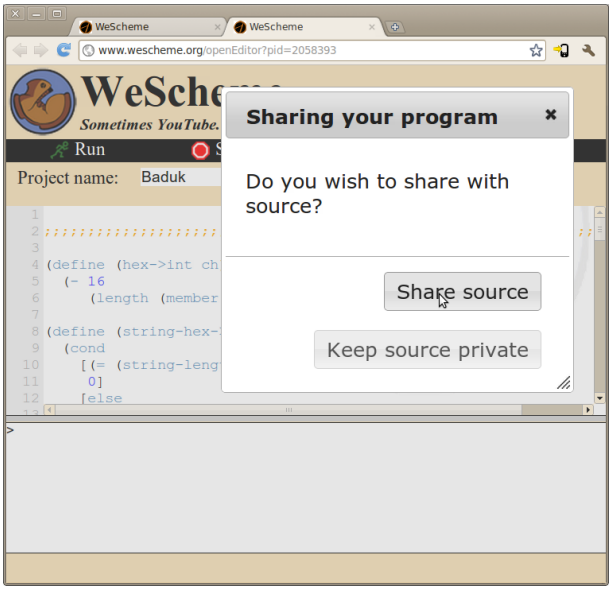


Figure 6.5 A Shared URL Link.

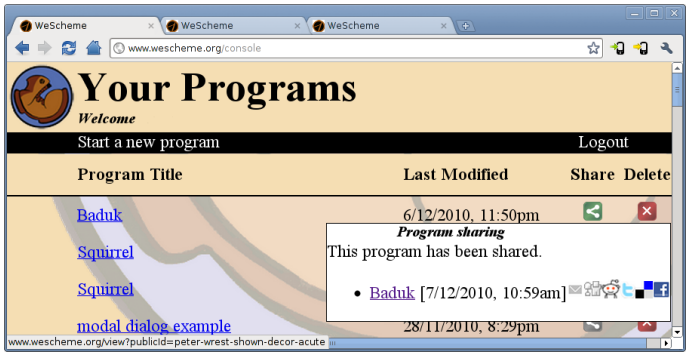


Figure 6.6 Visiting a Shared Program URL.

mentioned in Section 2.

Once the client receives the compiled output, the evaluator on the client's browser evaluates it and produces a value, which is finally displayed at the REPL. These values are converted into DOM nodes, either via explicit support (like strings and booleans) or by providing a `toDomNode()` method that the pretty printer can use to render the value. Some implementations of `toDomNode()` use the graphical and dynamic features of HTML 5 and JavaScript: an image value uses a `canvas`, and a rational number uses either `sup` and `sub` to produce a fraction or a repeated decimal expansion representation, switchable by mouse-clicking the representation.

By virtue of exploiting the browser, the environment immediately and automatically inherits improvements made by browser implementers. For instance, Web browsers recently added support for embedding videos inside Web pages in preparation for HTML 5. For a user to include a video in the output page (for instance, as a backdrop to a game) required no additional work. To make these videos programmatic objects—so that the user could, for instance, query the video or send it commands—required only a small amount of wrapping to make it an object in the evaluator, of the order of about ten lines of code, most of which are boilerplate.

The programs that users write are automatically backed up to and saved on the Cloud. WeScheme currently uses Google's AppEngine for this purpose. AppEngine also provides the web hosting of the online editor and the WeScheme user interface, which allows the system to scale in response to load.

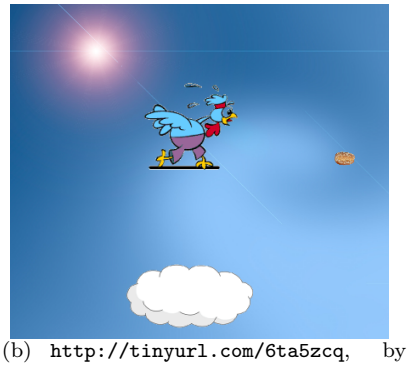
Of course, this architecture assumes continuous access to a server. This assumption is common in many contemporary Web-based systems such as Yahoo! Mail, Google Maps, etc. However, there is no difficulty in generating a binary that can be installed on the local host that provides access to the compiler without the need for networking support. In this setup WeScheme would still run in the browser, but to initiate it the user would connect to a URL on the local machine rather than to `www.wescheme.org`. A launcher could automate this process by automatically feeding the URL to the browser, so that the details are sufficiently transparent to the user. The architecture of the server compilation is stateless, so that it should be straightforward to load-balance requests across several compilation servers.

6.1.2 Usage and experience

WeScheme is in active use by students in Bootstrap [37], an afterschool program that teaches algebra through programming. Students have created over 2000 programs in WeScheme, and over a quarter of these have been shared, most of them with the source made public. (These numbers naturally change on a daily basis.) Figure 6.7 shows two games written by Bootstrap students.

Figure 6.7 Examples of games written by Bootstrap students, running under WeScheme.

(a) `tinyurl.com/cmtxem7`, by Jason and Leanna.



(b) `http://tinyurl.com/6ta5zcq`, by Paramita Roy and Sydney Lyte.

Many WeScheme users teach at schools with extremely limited computing infrastructure. As mentioned in Section 1, users face two different kinds of limitations: limited computing power, and locked-down systems. When systems are locked down, it becomes impossible to install a programming environment like DrRacket, and when the systems are weak, DrRacket's resource consumption is so great that just starting and running it is virtually impossible. In contrast, these machines can still run Web browsers with WeScheme successfully.

For instance, a collaborator recently had a school running Pentium III hardware and Windows 2000 software, on 256 Mb of RAM—a configuration over a decade old as of this writing. On such a system today's DrRacket will barely start, but students were able to write and run modest programs in WeScheme. As browsers get leaner and more efficient, this gives WeScheme a significant engineering edge.

WeScheme's user interface is intentionally spartan, in contrast to the visual complexity of many contemporary programming environments. In this regard, and in many details, WeScheme mimics DrRacket. The differences, however, suggest ways in which DrRacket can improve. The use of certain Web-specific metaphors allow WeScheme to take advantage of the user interfaces that users have already been trained to use. The presentation of error messages in WeScheme, for example, diverges from DrRacket: Web hyperlinks present error reports and also stack traces. In contrast, DrRacket uses a icon, which users must click on, to represent stack traces. Many students have reported confusion about what that DrRacket icon represents, and do not even know that it is clickable. In contrast, students have no difficulty understanding the visual metaphor of the hyperlink, and indeed it invites their exploration.

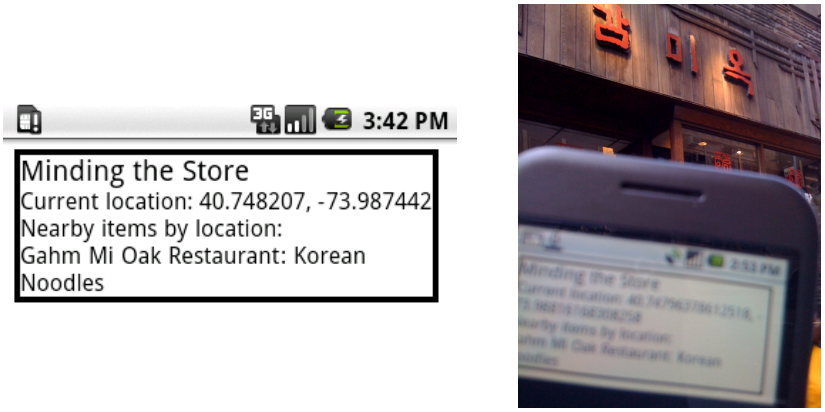
Because the programs are on the Web, readers can access them easily. Readers can both run, and view the source of (and thus easily modify and create their own versions of) any shared program. For example, `tinyurl.com/2924s2s` presents a game, while `tinyurl.com/28jptyn` shows a use of the browser's display framework; both URLs allow a user to run the program or inspect the program's source.

6.2 Moby

The Moby smartphone toolchain uses the same implementation strategy as WeScheme, taking advantage of the JavaScript-enabled web browser in smartphones. Many of the features that are phone-specific, such as geolocation, can already be accessed from the phone's JavaScript API, and therefore be sources of events for the World programming model. Figure 6.8 shows an example of a running `web-world` program that uses the user's latitude and longitude to note nearby landmarks. A middleware layer called PhoneGap [33] extends the phone's browser to include features that aren't currently accessible from the native web browser application, such as access to the tilt sensors. Moby builds phone application packages by combining a customized version of the phone's web browser with the Phonegap middleware layer.

One of the factors driving improvement in browsers is the use of the same core browser engines in mobile platforms. Indeed, smartphone Web browsers are now sophisticated enough that one can run WeScheme

Figure 6.8 An example of a Geolocation-aware phone program.



directly in the phone, though of course using the editor is an exercise in masochism. However, while the phone is a poor *editing* medium, it is perfectly reasonable to *run* programs on phones. Though some phone manufacturers lock down application stores and place limits on choices of programming languages, they still allow the deployment of applications using JavaScript in browsers. As phones are increasingly taken seriously as computing platforms, and browsers are recognized as an important component, the decision to target JavaScript, which may have seemed idiosyncratic, makes sense. The compiler underlying WeScheme has also been used in college-level courses that produce mobile phone applications.

Chapter 7

Conclusion

The need for a programming environment that could support the educational goals of the Bootstrap project motivated this work. Bootstrap is a curriculum that teaches geometric and algebraic concepts through the medium of computer programming, and it depends on the availability of an accessible, functional programming environment that supports World programming. Although DrRacket can support most of these requirements, logistic issues hamper its use: issues with software installation and low-powered machines on school computers, coupled with a restrictive anchoring to the desktop, can reduce the availability of the curriculum. A satisfactory solution demands a programming environment that can run on the Web. This dissertation fills this need, and makes technical and pedagogic contributions toward that end.

7.1 Technical contributions

The work in this dissertation enables event-driven functional programming on the Web by the design and implementation of a compiler and runtime for the Racket language which runs on top of JavaScript. It includes an implementation of the World programming model that has been extended to take advantage of web browser features. In the context of JavaScript that runs on web browsers, the evaluator needs to deal with asynchronicity and the capture, release, and reinstatement of control to maintain reliable interactions with the browser. Miscellaneous issues, such as tail calls and unbound recursion, must also be managed.

- We contribute a compiler and runtime environment that allows the evaluation of functional programs, and adapts asynchronous JavaScript APIs to act functionally.
- We contribute a foreign function interface (FFI) that allows advanced programmers to expose low-level JavaScript APIs as functional APIs. This FFI can used to connect with external libraries and Web services. These asynchronous APIs can be exposed in two ways: (1) a synchronous interface that offers a functional, blocking function, and (2) an event-driven World interface.
- We extend the World model to a *web-world* model that encompasses the browser DOM. We show that an approach that treats the DOM like a canvas, as something that can be computed as a function of the world value alone, has insufficient expressive power for simple kinds of World programs for the Web. By forcing the user to manage user interface state explicitly in the world, such a model inhibits composition with other programs running concurrently with the same DOM. The *web-world* extension treats the DOM as an additional kind of managed state, separate from the world value. It provides a zipper API that allows the DOM to be manipulated functionally but with much of the convenience of an imperative interface.
- We demonstrate the applicability the multiple-value-return technique of Ashley and Dybvig [4] in the context of JavaScript though a somewhat unusual use of function objects and its attributes to simulate address arithmetic.

7.2 Pedagogic contributions

Pedagogically, we contribute the WeScheme programming environment. It borrows heavily from the design of DrRacket, and present a simplified programming environment. It offers interactive evaluation in the browser as well a programming text editor with syntax highlighting and indentation, with proper source-specific highlighting on error messages. The language supports the linguistic forms of Beginner Student Language used by Racket.

Unlike DrRacket, WeScheme includes facilities for sharing programs, and provides APIs that are available only on the Web. Furthermore, all programs hosted on WeScheme can be deployed on a standard web browser, including those on smartphones; such phone-deployed programs can access phone-specific inputs and outputs through the World framework. At the time of this writing, it has hundreds of users, and more than 2000 programs have been shared.

Now that the WeScheme environment has been developed, it can provide a platform for language experiments. Guillaume Marceau [29] [30] shows that the content of error messages has an enormous impact on beginners. WeScheme can serve as a test bench for measuring the effectiveness of error messages targetted to novice programmers. One experiment may test the hypothesis that certain terms are easier for beginners than others, by using different sets of terms for error message content. Another experiment may test the hypothesis that error messages can actively teach terminology by using fine-grained colored highlighting that correlates technical terms to their referents in the source program. These kinds of user studies can be performed seamlessly because WeScheme is a central hub that can transparently record all user interactions. Similarly, it should be possible to expand WeScheme's scope to be more than just a program evaluator and code repository, to also include content such as worksheets and tutorials.

7.3 Future work

Some of the features of DrRacket have not yet been replicated in WeScheme. WeScheme does not currently support the Check Syntax functionality provided in DrRacket, which is a lightweight analysis tool that lets a user inspect where variables are defined and used. WeScheme also does not currently include support for graphical syntax, which allows program source to be expressed using graphical user interfaces rather than pure text. These both require extra-linguistic support from the program editor; because the program editor is an external component (CodeMirror), such support was out of scope of the dissertation. Also, WeScheme does not currently implement an algebraic stepper: although the language supports the underlying continuation-marks infrastructure, DrRacket's Stepper itself is currently being redesigned, making efforts toward re-implementing the stepper wasteful.

Another avenue for future work involves extending the World programming model to naturally encompass *impulsive* side effects. A missing element in the World programming model is to be able to use outputs that depend on time. For example, images can be displayed repeatedly without any harmful effect, but the playing of a sound is not an idempotent operation: it depends on time. It would be useful to find the right abstractions that can expose side effects in a way that fits naturally with World programing style.

On the technical side: the engineering effort expended to implement the most essential parts of the runtime motivates a study into what really should constitute the kernel of a programming language. Early in the implementation stages, it became clear that Racket is a huge language: despite its deceptive simplicity, it includes more than a thousand primitive operations, running the gamut from simple data manipulation to operating system responsibilities. As of this writing, a substantial portion of the Racket primitive kernel library has not been implemented in our runtime. It is future work to complete the runtime to cover these primitives. However, it would also be worthwhile to see if the Racket primitive standard library could be reduced, so that future implementors can more easily retarget Racket to alternative platforms.

There is low-hanging fruit that can be used to optimize the output of the compiler. For example, by analyzing a function's body, the compiler may be able to can guarantee that it does not apply unbounded recursion nor involve continuation capture; in this case, the compiler may employ a simpler compilation strategy that uses the native JavaScript function calling conventions.

Appendices

Appendix A

Compiler

The figures in this section show how the compiler generates Javascript* sequences for several representative Racket bytecodes.

Figure A.1 Compiling a branch.

```
(define (compile-branch exp cenv target linkage)
  (let ([f-branch: (gensym 'falseBranch)]
        [after-if: (gensym 'afterIf)])
    (let ([consequent-linkage
           (cond
            [(NextLinkage? linkage)
             (let ([value-context (NextLinkage-context linkage)])
               (make-LabelLinkage after-if: value-context))]
            [(ReturnLinkage? linkage)
             linkage]
            [(LabelLinkage? linkage)
             linkage])])
      (let ([p-code (compile (Branch-predicate exp) cenv 'val NEXT-LINKAGE/EXPECTS-SINGLE)]
            [c-code (compile (Branch-consequent exp) cenv target consequent-linkage)]
            [a-code (compile (Branch-alternative exp) cenv target linkage)])
        (append-instruction-sequences
         p-code
         (make-TestAndJump "MACHINE.val===false" f-branch:)
         c-code
         f-branch: a-code
         (if (NextLinkage? linkage) after-if: '()))))))
```

Figure A.2 Compiling `WithContMark`

```
;; compile-with-cont-mark: WithContMark CompileTimeEnvironment Target Linkage → InstructionSequence
(define (compile-with-cont-mark exp cenv target linkage)

  ;; in-return-context: → InstructionSequence
  (define (in-return-context)
    (append-instruction-sequences
     (compile (WithContMark-key exp) cenv 'val NEXT-LINKAGE/EXPECTS-SINGLE)
     (make-Assign (make-ControlFrameTemporary 'pendingContinuationMarkKey)
                  (make-Reg 'val))
     (compile (WithContMark-value exp) cenv 'val NEXT-LINKAGE/EXPECTS-SINGLE)
     (make-InstallContinuationMarkEntry!)
     (compile (WithContMark-body exp) cenv target linkage)))

  ;; in-non-tail-context: (U NextLinkage LabelLinkage) → InstructionSequence
  (define (in-non-tail-context linkage)
    (let ([on-return/multiple: (gensym 'procReturnMultiple)]
          [on-return: (make-LinkedLabel (gensym 'procReturn) on-return/multiple:)])
      (append-instruction-sequences
       ;; Making a fresh frame to record the key/value:
       (make-PushControlFrame on-return)
       (compile (WithContMark-key exp) cenv 'val NEXT-LINKAGE/EXPECTS-SINGLE)
       (make-Assign (make-ControlFrameTemporary 'pendingContinuationMarkKey)
                    (make-Reg 'val))
       (compile (WithContMark-value exp) cenv 'val NEXT-LINKAGE/EXPECTS-SINGLE)
       (make-InstallContinuationMarkEntry!)
       (compile (WithContMark-body exp) cenv 'val RETURN-LINKAGE/NON-TAIL)
       on-return/multiple:
         ;; code omitted; see source code in repository.
       on-return:
         ;; code omitted; see source code in repository.
       )))
  (cond
    [(ReturnLinkage? linkage)
     (in-return-context)]
    [(NextLinkage? linkage)
     (in-non-tail-context linkage)]
    [(LabelLinkage? linkage)
     (append-instruction-sequences
      (in-non-tail-context linkage)
      (make-Goto (make-Label (LabelLinkage-label linkage))))]))
```

Figure A.3 Compiling App, phase 1: evaluating the procedure and its operands.

```
;; compile-application: App CompileTimeEnvironment Target Linkage → InstructionSequence
(define (compile-application exp cenv target linkage)
  (let* ([extended-cenv
          (extend-compile-time-environment/scratch-space
           cenv
           (length (App-operands exp)))]
         [proc-code (compile (App-operator exp)
                             extended-cenv
                             (if (empty? (App-operands exp))
                                 'proc
                                 (make-EnvLexicalReference
                                  (sub1 (length (App-operands exp)))
                                  #f))
                             NEXT-LINKAGE/EXPECTS-SINGLE)]
         [operand-codes (map (lambda (operand target)
                               (compile operand
                                        extended-cenv
                                        target
                                        NEXT-LINKAGE/EXPECTS-SINGLE))
                             (App-operands exp)
                             (build-list (length (App-operands exp))
                                         (lambda (i)
                                           (if (< i (sub1 (length (App-operands exp))))
                                               (make-EnvLexicalReference i #f)
                                               'val))))])
    (append-instruction-sequences
     (make-PushEnvironment (length (App-operands exp)) #f)
     proc-code
     (apply append-instruction-sequences operand-codes)
     (if (empty? operand-codes)
         empty-instruction-sequence
         (append-instruction-sequences
          (make-Assign 'proc (make-EnvLexicalReference n #f))
          (make-Assign (make-EnvLexicalReference n #f) (make-Reg 'val))))
     (make-Assign 'argcount (make-Const (length (App-operands exp))))
     (compile-general-procedure-call cenv
                                     (make-Const (length (App-operands exp)))
                                     target
                                     linkage))))
```

Figure A.4 Compiling `App`, phase 2: transferring control and establishing return points in non-tail contexts.

```
;; compile-general-procedure-call: CompileTimeEnvironment Natural Target Linkage → InstructionSequence
(define (compile-general-procedure-call cenv number-of-arguments target linkage)
  (let* ([entry-point-target
          (make-CompiledProcedureEntry (make-Reg 'proc))])
         [on-return/multiple: (gensym 'procReturnMultiple)]
         [on-return: (make-LinkedLabel (gensym 'procReturn) on-return/multiple:)])

    (cond [(ReturnLinkage? linkage)
           (cond
             [(eq? target 'val)
              (cond
               [(ReturnLinkage-tail? linkage)
                (append-instruction-sequences
                 (make-PopEnvironment (make-Const (length cenv)) number-of-arguments)
                 (make-SetFrameCallee! (make-Reg 'proc))
                 (make-Goto entry-point-target))]
               [else
                (make-Goto entry-point-target)])]
             [else
              (error 'compile "return linkage, target not val: ~s" target)])]

          [(or (NextLinkage? linkage) (LabelLinkage? linkage))
           (let* ([nontail-jump-into-procedure
                   (append-instruction-sequences
                    (make-PushControlFrame/Call on-return: )
                    (make-Goto entry-point-target))]
                  [check-values-context-on-procedure-return
                   ;; this is code to check context for
                   ;; correct number of arguments.
                   ;; See Figure A.5.
                   )]
                  [maybe-migrate-val-to-target
                   (cond
                    [(eq? target 'val)
                     empty-instruction-sequence]
                    [else
                     (make-Assign target (make-Reg 'val))])]
                  [maybe-jump-to-label
                   (if (LabelLinkage? linkage)
                       (make-Goto (make-Label (LabelLinkage-label linkage)))
                       empty-instruction-sequence)])

             (append-instruction-sequences
              nontail-jump-into-procedure
              check-values-context-on-procedure-return
              maybe-migrate-val-to-target
              maybe-jump-to-label))])))
```

Figure A.5 Code that generates the multiple-value context checks for a procedure's return to a non-tail context. Each case must generate code that involve both *on-return/multiple:* and *on-return:* labels, to dynamically handle returns with either multiple or single values.

```
(cond
  ;; case 1: indifferent to multiple results
  [(eq? context 'drop-multiple)
   (append-instruction-sequences
    on-return/multiple:
    (make-PopEnvironment (new-SubtractArg (make-Reg 'argcount) (make-Const 1))
                         (make-Const 0))
    on-return: )]

  ;; case 2: expecting multiple results
  [(eq? context 'keep-multiple)
   (let ([after-return: (gensym 'afterReturn)])
     (append-instruction-sequences
      on-return/multiple:
      (make-Goto (make-Label after-return: ))
      on-return:
      (make-Assign 'argcount (make-Const 1))
      after-return: ))]

  ;; case 3: expecting an exact number of results
  [(natural? context)
   (cond
     [(= context 1)
      (append-instruction-sequences
       on-return/multiple:
       (make-RaiseContextExpectedValuesError! 1)
       on-return: )]
     [else
      (let ([after-value-check: (gensym 'afterValueCheck)])
        (append-instruction-sequences

          ;; if the wrong number of arguments come in, die
          on-return/multiple:
          (make-TestAndJump (format "(MACHINE.argcount-~a)===0" context)
                            after-value-check: )
          on-return:
          ;; If we return with a single value, then we know we've messed up
          ;; since we expected context ¡¿ 1.
          (make-RaiseContextExpectedValuesError! context)
          after-value-check: ))])])
```

Figure A.6 An implementation of Figure 5.15 using a distinguished structure.

```
var I;                  // global register i.
var STACK;              // global stack register.
var controlStack = []; // global control stack

var MultipleValueBox = function(v1, v2) { this.v1 = v1; this.v2 = v2; };
THE_MULTIPLE_VALUE = new MultipleValueBox(undefined, undefined);

var stackPush = function(aStack, elt) {
    controlStack.pop()(cons(elt, aStack));
};

var stackPop = function(aStack) {
    var result = THE_MULTIPLE_VALUE;
    result.v1 = car(aStack);
    result.v2 = cdr(aStack);
    controlStack.pop()(result);
};

var stackIsEmpty = function(aStack) {
    controlStack.pop()(aStack === EMPTY);
};

var test = function() {
    I = 0; STACK = EMPTY;
    loop1();
};

var loop1 = function() {
    if (I >= LIMIT) {
        loop2();
    } else {
        controlStack.push(afterPush);
        stackPush(STACK, I);
    }
};

var afterPush = function(v) {
    if (v === THE_MULTIPLE_VALUE)
        throw new Error("expected single result");
    I++;
    STACK = v;
    loop1();
};

var loop2 = function() {
    controlStack.push(afterCheckEmpty);
    stackIsEmpty(STACK);
};

var afterCheckEmpty = function(v) {
    if (v === THE_MULTIPLE_VALUE)
        throw new Error("expected one result");
    if (v) { alert("done!"); return; }
    controlStack.push(afterPop);
    stackPop(STACK);
};

var afterPop = function(v) {
    if (v === THE_MULTIPLE_VALUE) {
        STACK = v.v2;
        loop2();
    } else throw new Error("expected two results");
};
```

Figure A.7 An implementation of Figure 5.15 using a pair of continuations.

```
var I;                  // global register i.
var STACK;              // global stack register.
var controlStack = []; // global control stack

var stackPush = function(aStack, elt) {
    var kSingle = controlStack.pop(); // Note that each return point needs
    controlStack.pop();               // to account for both elements in controlStack.
    kSingle(cons(elt, aStack));
};

var stackPop = function(aStack) {
    controlStack.pop();
    var kMultiple = controlStack.pop();
    kMultiple(car(aStack), cdr(aStack));
};

var stackIsEmpty = function(aStack) {
    var kSingle = controlStack.pop();
    controlStack.pop();
    kSingle(aStack === EMPTY);
};

var test = function() {
    I = 0; STACK = EMPTY; loop1();
};

var loop1 = function() {
    if (I >= LIMIT) {
        loop2();
    } else {
        controlStack.push(errorOnMultiple);  // In this use of afterPush, we want to error
        controlStack.push(afterPush);        // when we receive two values.
        stackPush(STACK, I);
    }
};

var afterPush = function(v) {
    I++;
    STACK = v;
    loop1();
};

var errorOnMultiple = function(v1, v2) { throw new Error("Expecting single result"); };

var loop2 = function() {
    controlStack.push(errorOnMultiple);
    controlStack.push(afterCheckEmpty);
    stackIsEmpty(STACK);
};

var afterCheckEmpty = function(v) {
    if (v) { alert("done!"); return; }
    controlStack.push(afterPop);
    controlStack.push(errorOnSingle);
    stackPop(STACK);
};

var errorOnSingle = function(v) { throw new Error("Expecting two results"); };

var afterPop = function(v1, v2) {
    STACK = v2;
    loop2();
};
```

Figure A.8 An implementation of Figure 5.15 using an attribute on the function object to hold the multiple-value context.

```
var I;                  // global register i.
var STACK;              // global stack register.
var controlStack = []; // global control stack

var stackPush = function(aStack, elt) {
    // Functions that return single values just call the continuation
    // as usual:
    controlStack.pop()(cons(elt, aStack));
};

var stackPop = function(aStack) {
    // Functions that return multiple values call the 'multiple'
    // continuation.
    controlStack.pop().multiple(car(aStack), cdr(aStack));
};

var stackIsEmpty = function(aStack) {
    controlStack.pop()(aStack === EMPTY);
};

var test = function() {
    I = 0; STACK = EMPTY; loop1();
};

var loop1 = function() {
    if (I >= LIMIT) {
        loop2();
    } else {
        controlStack.push(afterPush);
        stackPush(STACK, I);
    }
};

var afterPush = function(v) {
    I++;
    STACK = v;
    loop1();
};
afterPush.multiple = function(v1, v2) { throw new Error("Expecting single result"); };

var loop2 = function() {
    controlStack.push(afterCheckEmpty);
    stackIsEmpty(STACK);
};

var afterCheckEmpty = function(v) {
    if (v) { alert("done!"); return; }
    controlStack.push(afterPop);
    stackPop(STACK);
};
afterCheckEmpty.multiple = function(v1, v2) { throw new Error("Expecting single result"); };

var afterPop = function(v) { throw new Error("expected two results"); };
afterPop.multiple = function(v1, v2) {
    STACK = v2;
    loop2();
};
```

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Intepretation of Computer Programs*. MIT Press, second edition, 1996.

[2] Peter Achten and Rinus Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5, 1995.

[3] Ken Anderson, Tim Hickey, and Peter Norvig. JScheme. `jscheme.sourceforge.net/`.

[4] J. Michael Ashley and R. Kent Dybvig. An Efficient Implementation of Multiple Return Values in Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 1994.

[5] Brenda S. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 1977.

[6] James R. Bell. Threaded code. *Communications of the ACM*, 1973.

[7] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object Oriented Languages and Programming Systems*, 2009.

[8] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. In *ACM Translations on Programming Languages and Systems*, 2004.

[9] Matthias Felleisen, Robert Bruce Findler, Kathi Fisler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Worlds*. `world.cs.brown.edu`, 2008.

[10] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A Functional I/O System or, Fun for Freshman Kids. *International Conference on Functional Programming*, 2009.

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, second edition, 2010. `www.ccs.neu.edu/home/matthias/HtDP2e/`.

[12] Robert Bruce Findler and PLT. DrRacket: Programming Environment. Technical Report PLT-TR-2010-2, PLT Inc., 2010. `racket-lang.org/tr2/`.

[13] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `racket-lang.org/tr1/`.

[14] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. *International Conference on Functional Programming*, 2007.

[15] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1985.

[16] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *Programming Language Design and Implementation*, 2009.

[17] Deniz A. Gürsel, Uğur Çekmez, and R. Emre Başar. Implementation of Scheme Numeric System for JavaScript. *Akademik Bilişim Konferanslari*, 2010.

[18] Gérard Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 1997.

[19] Google Inc. Google Maps API. `https://developers.google.com/maps/`.

[20] JQuery.com. Jquery. `jquery.com`.

[21] JUnit.org. Junit. `www.junit.org`.

[22] Victor J. Katz. Algebra: Gateway to a Technological Future. Technical report, The Mathematical Association of America, 2007.

[23] Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket Virtual Machine and Randomized Testing. Technical report, Northwestern University, 2010. `plt.eecs.northwestern.edu/racket-machine/`.

[24] Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. Lively Fabrik: A Web-based End-user Programming Environment. In *Creating, Connecting and Collaborating through Computing*, 2009.

[25] Florian Loitsch. Exceptional Continuations in JavaScript. In *Workshop on Scheme and Functional Programming*, 2007.

[26] Florian Loitsch and Manuel Serrano. Hop Client-Side Compilation. In *Trends in Functional Programming*, 2007.

[27] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: A Sneak Preview. *International Conference on Creating, Connecting and Collaborating through Computing*, 2004.

[28] John H. Maloney and Randall B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. *User Interface Software and Technology*, 1995.

[29] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind Your Language: On Novices' Interactions with Error Messages. *SPLASH/Onward!*, 2004.

[30] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. *Technical Symposium on Computer Science Education*, 2011.

[31] Google Inc. Google App Inventor. `appinventor.googlelabs.com`.

[32] Google Inc. Google Web Toolkit. `code.google.com/webtoolkit/`.

[33] Nitobi Software Inc. PhoneGap. `www.phonegap.com`.

[34] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *Proceedings of the 2009 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2009.

[35] Lyle Ramshaw. Eliminating go to's while Preserving Program Structure. *Journal of the ACM*, 1988.

[36] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn. JSMeter: Measuring JavaScript Behavior in the Wild. Technical Report MSR-TR-2010-8, Microsoft Research, 2010. `research.microsoft.com/apps/pubs/?id=118663`.

[37] Emmanuel Schanzer. Bootstrap. `www.bootstrapworld.org`.

[38] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. *Revised*^{6} *Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[39] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic Native Optimization of Interpreters. *Proceedings of the 2003 workshop on Interpreters, virtual machines, and emulators*, 2003.

[40] Antero Taivalsaari, Tommy Mikkonen, Dan Ingalls, and Krzysztof Palacz. Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, Oracle, 2008. `labs.oracle.com/techrep/2008/abstract-175.html`.

[41] W3C. GeoLocation API Specification. `dev.w3.org/geo/api/spec-source.html`.

[42] Tom Wu. JSBN: BigIntegers and RSA in JavaScript. `www-cs-students.stanford.edu/~tjw/jsbn`.

[43] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of Dynamic Languages using Hierarchical Layering of Virtual Machines. *Dynamic Languages Symposium*, 2009.

[44] Alon Zakai. Emscripten: an LLVM-to-JavaScript Compiler. `code.google.com/p/emscripten/`.

[45] Nicholas C. Zakas. JavaScript stack overflow error. `www.nczonline.net/blog/2009/05/19/javascript-stack-overflow-error/`.