Whalesong: Running Racket in the Browser

Danny Yoo
WPI, University of Utah
dyoo@hashcollision.org

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

JavaScript is the language of the ubiquitous Web, but it only poorly supports event-driven functional programs due to its single-threaded, asynchronous nature and lack of rich control flow operators. We present Whalesong, a compiler from Racket that generates JavaScript code that masks these problems. We discuss the implementation strategy using delimited continuations, an interface to the DOM, and an FFI for adapting JavaScript libraries to add new platform-dependent reactive features. In the process, we also describe extensions to Racket's functional event-driven programming model. We also briefly discuss the implementation details.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords Racket, Web, browsers, JavaScript

1. Introduction

Racket is a widely-used programming language, especially in educational contexts. In particular, Racket is the basis for the Bootstrap project [17], which uses the functional subset of Racket to teach computer science and algebra together to students from middle-school onward. The curriculum combines functions with a functional I/O model [9] to teach testing-friendly event-driven programming 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.

For various reasons, it is important for these programs to run in the browser. First, it makes it possible for students to share the fruits of their labor with family and friends. Second, many schools have locked-down computer systems so that installing new run-time systems is difficult or impossible. (An earlier paper describes our cloud-based programming environment, WeScheme [20].) Finally, students want to be able to run their games on platforms other than the desktop, such as tablets and mobile phones; the ubiquity of browsers saves us from having to port Racket these platforms. Thus, translating to JavaScript lowers barriers, and increases opportunities, in education.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DLS '13, October 28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2433-5/13/10…$15.00.
http://dx.doi.org/10.1145/2508168.2508172

However, it is not straightforward to compile a language like Racket to JavaScript, despite their superficial similarities. The JavaScript run-time poses several crucial obstacles:

1. JavaScript's single-threaded execution model does not easily enable computations to be preempted or interrupted, making it difficult to guard the programming environment's behavior against out-of-control computations (so that the programming environment can, for instance, provide a "Stop" button).
2. JavaScript programs written in functional style can hit runtime limitations with regards to the JavaScript stack ceiling. This is exacerbated by the lack of cross-platform support for tail-calls in JavaScript.
3. The asynchronicity of many JavaScript APIs poses difficulties with a functional programming style because values may be produced that are not yet fully initialized.

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

Contributions This paper presents Whalesong, a compiler that enables Racket programs to run in JavaScript, and thus on the Web. This provides features not supported by JavaScript, such as interrupts and cooperative timesharing. Though Whalesong supports the Racket language [11], including its imperative, object-oriented, and other parts, we focus here on the functional parts, since these are actually the most interesting. This paper describes how Whalesong extends a functional event-driven programming model; implements these in the hostile JavaScript environment; offers an interface to the DOM; and provides a foreign-function interface for interacting with JavaScript libraries. We also discuss the implementation and provide some performance details. Finally, Whalesong also provides a large run-time system to port Racket programming primitives, including the numeric tower. This paper ignores these significant but technically uninteresting aspects.

2. Background: The World Programming Model

We first introduce the World programming model [9], which is the basis for interactive programming in the Bootstrap curriculum and in several other curricula that use Racket. The model is also smoothly extensible: for instance, Whalesong provides handlers that automatically extend the model to respond to events on devices like mobile phones (such as the receipt of text messages, or the processing of GPS locations).

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 infor-

mation 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 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?*) consumes the world and performs 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 the run-time system renders on-screen. The *hits-floor?* predicate describes when the ball has reached the floor. Finally, the *big-bang* call in Figure 1 begins an event loop that uses the functions provided by *on-tick*, *to-draw*, and *stop-when* to drive its behavior.

In general, there are three classes of functions representing the reactive program:

- update: how the world changes
- output: how the world can be presented (e.g., on a screen)
- (optional) termination: how the computation can stop

In short, the World model is a functional instantiation of Model-View-Controller, 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. On other platforms, Whalesong supports additional updaters and effects: e.g., *on-tilt* to sense the orientation of a mobile phone or other orientation-sensing device is moved in three dimensions, and effects like emitting sounds or sending text messages in addition to drawing.

Because users write pure functions, even when responding to external state, they can exercise these functions in the REPL, and even write unit tests to ensure that their functions are producing valid results. Figure 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.

Figure 1 Event-driven animation of a descending ball

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

;; The world is a number (distance from the top of the screen).

(define RED-BALL (circle RADIUS "solid" "red"))

(define MID-WIDTH (quotient WIDTH 2))

;; 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.
(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, the height is zero
          (on-tick descend 1/15)   ;; ... 15 frames a second
          (to-draw draw)           ;; ... use render to draw the scene
          (stop-when hits-floor?))  ;; ... and stop when the ball hits the floor
```

big-bang as an expression The World framework does more than just simplify the event-handling boilerplate: it enables the reactive computation to be used in arbitrary expressions. When the function bound to *stop-when* evaluates to a true value, the event loop ends; its value is that of the last world before the loop ended, which can be used as the basis for a subsequent computation (e.g., it might represent a game's state, which the game uses to configure the initial state of the next round).

Although *big-bang* appears only as a top-level statement in Figure 1, 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. For instance, the world might represent a game's state, and each round might result in one event loop. When a level finishes, its *big-bang* returns the final state at that level, which can be used to configure the next level (e.g., starting with the right number of remaining lives) with just algebraic composition.

In fact, Whalesong's implementation of *big-bang* allows *big-bang* expressions to be *nested*. This goes beyond the (former) implementation in Racket itself in that each nested *big-bang* is treated as its own "virtual machine", i.e., a nested *big-bang* represents a *modal* event loop. Thus it becomes straightforward to implement yes/no prompts and other modal features in a program in the most natural way: by simply nesting *big-bang* expressions.[1]

Implementation Challenges Unfortunately, implementing this model in a browser's JavaScript is not trivial. The main obstacles lie in asynchronicity and event-driven event-loops:

- 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 Whalesong provides 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 when the image has finished loading.

  In fact, although the low-level image constructor produces a value, that value is not safe for use until the image is fully

[1] Racket's implementation still does not behave this way. Outer handlers continue to run while inner handlers are active, resulting in potentially confusing behavior.

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 order to receive any events from the browser, such as timer ticks or button presses, the main thread of evaluation needs to relinquish control 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 cannot return a useful value to the caller because it hasn't finished its computation yet. This violates the expectation 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.

  A programming environment faces a related problem: it should be possible to interrupt a program's execution so that users can curtail out-of-control programs. However, an evaluating program retains control, again preventing any user-interface events (such as a "Stop" button) from being processed until 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.

3. Implementation using Delimited Continuations

In contrast to JavaScript's limited control operators (function calls, exceptions), languages like Racket support *delimited continuations* [12] to provide mechanisms for non-local control flow. (Section 6 explains how we implement them in JavaScript.) 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 to the nearest prompt.

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

This granularity becomes useful when trying to maintain boundaries between subsystems. For example, the interactive REPL of the programming environment uses finalization code that evaluates after each expression, and although that code is in 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.

We adopt these continuation primitives and apply them toward the problems discussed in Section 2 as follows:

3.1 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.

3.2 *big-bang*

big-bang is handled similarly to Section 3.1, 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.

3.3 Exception Handling

The proper handing of exceptions also poses issues. 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 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. Each element of this external stack is a JavaScript object whose fields include `label`, `marks`, and `tag` attributes. The `label` attribute holds JavaScript function values as representations of return addresses in the 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 work 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

[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 2 A program that counts elapsed time

```
;; The world is a number (elapsed time).

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

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

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.

4. World Programming with the DOM

The World programming model described in section 2 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 program fragments from different sources.[3]

Let us 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. In a browser, correspondingly, such a function might produce a Web page. For example, the program in Figure 2 shows a counter that uses a Web-facing 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 our API: a function *dom-bind* that enables a World program to connect world-updating functions to the events of the DOM.

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

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 3 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.

[3] This entire section represents another extension to Racket's World model, which only supports stateless output, such as images that can be constructed as a whole and replaced in their entirety. The ideas of this section apply equally well to other stateful views, such as object-oriented GUIs, though we have not implemented them in such contexts.

Figure 3 A program that counts button presses

```
;; The world is a number (number of button-presses).

;; 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 A problematic World program

```
;; The world is a number (elapsed time).

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

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

Problems This approach appears to be effective when there is no state in the DOM, but the button-clicking example in Figure 3 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 is non-uniform, where some events are bound in *draw* and others in the call to *big-bang*.

In addition, there is a more serious weakness: 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.

As an example, consider the program in Figure 4, which displays a counter and allows a user to type input into a text field. This program does not behave as one might expect: the user can 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 is called, the new DOM tree constructed by *draw* is rendered by the browser; as the world does not store the content 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 5, 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

Figure 5 Revised version of the problematic program from Figure 4

```
;; The world is a number (elapsed time)
;; and text (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))
```

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 demonstrates the weakness with this approach: even though a 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 of treating DOM output as a pure function based only on the world value fails to address several problems:

- The rich statefulness of form elements forces 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. This state can be unwieldy to express, and needs to keep track of constantly upgrading features in Web technology.

  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 user's 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 state to the next.

  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. We have experimented with a tree-differencing algorithm, but it is difficult to implement efficiently and without creating surprises for users. Figure 6 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.

Incorporating Views 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. Therefore, our revised library:

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, described below.
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:

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

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

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

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

A view represents an internal reference into a tree that remembers its outer context. The API encourages navigation on identifier rather than by the raw tree structure. Some of 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.

This allows world callbacks to functionally query the state of DOM elements. The separation of the view allows our 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 function provided to *to-draw* takes the existing view as an argument:

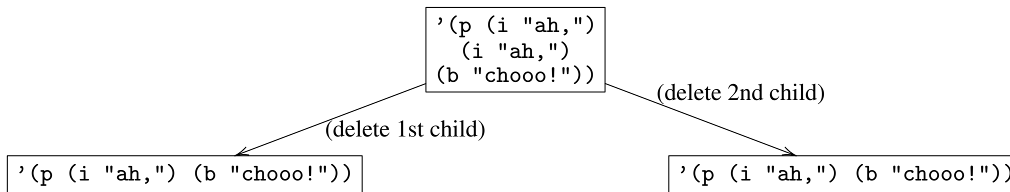
$$\textit{to-draw} : \textit{world} \rightarrow \textit{DOM} \quad \Longrightarrow \quad \textit{to-draw} : \textit{world}\ \boxed{\textit{view}} \rightarrow \boxed{\textit{view}}$$

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.

View Operations Generating a view should be as easy as generating an image. Looking back at the World model, a program generates images through additive operations (*place-image*, *overlay*, *beside*, etc.) that compose their arguments into larger images. 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, 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. Therefore, Whalesong

Figure 6 A example of an ambiguous patch due to tree-diffing: which effect was intended?



provides several view-manipulation operations that enable the user to construct and maintain cursors into the DOM tree, such as:

$$\begin{aligned}
\rightarrow\!\mathit{view} &: \mathit{dom\text{-}tree} \rightarrow \mathit{view} \\
\mathit{view\text{-}focus} &: \mathit{view}\ \mathit{string} \rightarrow \mathit{view} \\
\mathit{view\text{-}up} &: \mathit{view} \rightarrow \mathit{view} \\
\mathit{view\text{-}left} &: \mathit{view} \rightarrow \mathit{view} \\
\mathit{view\text{-}text} &: \mathit{view} \rightarrow \mathit{string} \\
\mathit{view\text{-}form\text{-}value} &: \mathit{view} \rightarrow \mathit{string}
\end{aligned}$$

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.

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

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

$$\mathit{view\text{-}bind} : \mathit{view}\ \mathit{event\text{-}type}\ \mathit{updater} \rightarrow \mathit{view}$$

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.

Examples Programs written with the view-based library are not much more verbose than their traditional World counterparts. Figure 7 shows the counting example (Figure 2). 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, 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 8. 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 user's program, it becomes trivial to query the state of elements in the view without having to pollute the world with extraneous detail. For instance, we can create a simple list-manager program that reads in an item from

Figure 7 Final version the clock-ticking program from Figure 2

```
;; The world is a number (elapsed time).

;; 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 8 Final version of the problematic program from Figure 4

```
;; The world is a number (elapsed time).

;; 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 1)
          (to-draw draw))
```

a text field whenever a button is pressed. This program is shown in Figure 9. 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]

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:

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

[4] *How to Design Programs*, second ed. [10], includes an 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 the text field's state.

Figure 9 A simple list-maker

```
;; The world is a list of strings (shopping list).

;; 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))
```

The first component is a tree zipper [13]. 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.

The second component of a view, 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, changing the text content of an element needs:

- focusing the view on the affected element,
- 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 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

Figure 10 Foreign-function interface creator

$$\begin{aligned}
\textit{js-function}{\rightarrow}\textit{proc} &: \textit{js-function} \rightarrow \textit{procedure}\\
\textit{js-async-function}{\rightarrow}\textit{proc} &: \textit{js-function} \rightarrow \textit{procedure}\\
\textit{make-world-event-handler} &: \textit{up-proc}\ \textit{down-proc} \rightarrow (\textit{updater} \rightarrow \textit{handler})\\
\textit{up-proc} &: \textit{js-function} \rightarrow X\\
\textit{down-proc} &: \textit{js-function}\ X \rightarrow \textit{void}
\end{aligned}$$

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 there is a deliberate discontinuity, and the DOM on the browser is discarded and replaced by the content in the zipper.

5. A Functional Foreign Function Interface

The *view-bind* function provides a simple mechanism to connect 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, the browser can notify programs when the physical location of the environment shifts to a different latitude and longitude. The API provides a type and functions to register and clear callbacks with the browser:

$$\textit{geo_cbk} : \{\ \textit{latitude} : \textit{double},\ \textit{longitude} : \textit{double}, \ldots\ \} \rightarrow \textit{void}$$

$$\textit{navigator.geolocation.watchPosition} : \textit{geo_cbk} \rightarrow \textit{watchId}$$

$$\textit{navigator.geolocation.clearWatch} : \textit{watchId} \rightarrow \textit{void}$$

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.

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 World programs to these APIs.

The FFI provides basic services to bind callbacks from JavaScript into Whalesong's Racket. It enables

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

Figure 10 shows a selection of the FFI. The library is intended to be thin, so it does no automatic coercion 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.

js-function→proc 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 procedure that can be used as a standard World event handler

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

Figure 11 Binding the GeoLocation API to World programs

```
;; start-up-geo: js-function → number
;; Initialize the JavaScript GeoLocation API.
(define start-up-geo
  (js-function->proc "
   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->proc
   "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))
```

(just like *on-tick* or *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 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*.

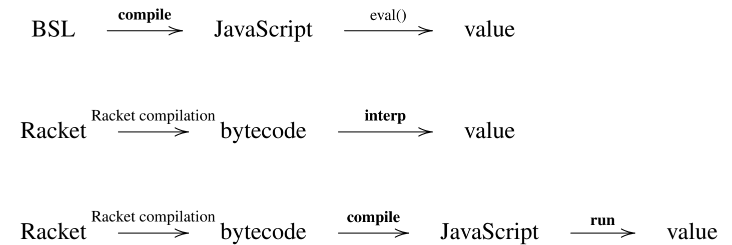
The code in Figure 11 demonstrates how the FFI can bind the GeoLocation APIs for use in World programs. The functions *start-up-geo* and *shut-down-geo* register a callback with the JavaScript environment, and the code uses these two functions to create a new World event handler called *on-geo-change*. 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 into the World event loop.

We have used this same API to bind to other services as well. For instance, in a similar amount of code to Figure 11, we have defined an interface to Google Maps. Once this is done, a client of this library can write World programs that interact with maps just as easily as they do built-in primitive types, using a *on-map-click* handler in *big-bang*.

6. Implementation

In section 3 we described problems with executing programs atop JavaScript, and the use of delimited continuations to circumvent them. Of course, JavaScript does not itself provide these primitives. We obtain them by implementing the full Racket language in JavaScript.

Figure 12 Summary of evaluator designs



Over the past five years, we have explored three different designs for the evaluator (Figure 12). The first design, which is self-evident, supports programs poorly because of JavaScript asynchrony issues and stack ceiling collisions, and is limited to a small language subset (BSL is the Beginning Student Language of DrRacket) due to the complexity of compiling all of Racket.

The second version extends the language by employing Racket itself to perform the compilation.[6] Racket generates programs in a well-defined bytecode language. These bytecodes are executed by an interpreter written in JavaScript.

The inner loop of the interpreter dispatches on the type of the bytecode operations, with a subroutine threading approach [5] 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 enables the implementation of control operators, continuation marks [8], and more, but suffers from very poor performance—especially on smart-phones and tablets—for anything beyond basic animations and programs. Programs run under this system take 1,000-10,000 times what they do in Racket!

Current Design The basic problem with the second design is that we have introduced an interpreter loop. This not only reintroduces the overhead of instruction dispatch that the compilation step tried to eliminate, it also creates programs that especially obscure their control flow from a JIT. If the JIT provides hooks for the language implementor to provide runtime hints to expose logical loop structure, to virtualize the program counter, then the JIT can perform much better [6, 18, 19]. However, these hinting mechanisms have not yet been incorporated into mainstream JavaScript evaluators.

Therefore, our current design melds the previous two designs. Like the second design, it reuses the Racket compiler to handle details of macro expansion and linguistic support. Furthermore, it explicitly manages the control context through a trampoline (**run**) that also implements control operators. Like the first design, it

[6] Because the Racket compiler is not entirely self-hosting, for now we run this compiler on a cloud server (though in principle it could be run by a native browser extension, a local server app, etc.)—though this means that programs that perversely use reflection to cross compilation phases can detect a difference between the platforms. On the positive side, many languages—including Algol 60, Python, etc.—have been implemented in Racket, so this approach makes it possible to host many more languages than Racket itself. In particular, we are using Whalesong to host a new, non-parenthetical language called Pyret.

eliminates the cost of interpretation by using a **compile** process—in the register-machine style of SICP [1]—to generate JavaScript *extended* with explicit GOTO statements and labels, which are then desugared into regular JavaScript.

There are two basic ways of simulating GOTO statements:

- 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. The entry point of each transformed block manages the maximum height of the stack. A trampoline 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.

Figure 13 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.

However, the case-switch is applicable only if all the basic blocks are present at compilation time. In contrast, the trampolining approach can be applied in dynamic linking situations, such as that of interactive evaluation and dynamic module loading. 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 **run** provides the necessary support for calling them.[7]

6.1 The Trampoline

All evaluation happens in the context of a top-level trampoline. The trampoline starts by calling a JavaScript function within a `try/catch`; each function constructed by the Racket-to-JavaScript compiler consumes the virtual machine as its sole argument, and checks the trampoline register. If this value reaches zero, the function `throws` itself as an exception back to the trampoline. The trampoline may then either restart the computation, or use `setTimeout` to schedule the current computation for later, and switch to another one or yield control to the browser. In either case, the `throw` allows the runtime to discard stack frames on the native JavaScript stack. The top-level exception handler monitors other exception types to implement operations like continuation capture or to translate low-level JavaScript errors into Racket errors.

6.2 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 16 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 also rife with asynchrony. Although the example in Figure 16 is artificial, the fundamental problem exists, and users of Whalesong 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, we have had to simulate locks in JavaScript. Figure 17 shows **run**'s implementation of mutual exclusive locks; note that `this.locked` can be read and written in two separate statements (rather than needing an atomic compare-and-swap) because JavaScript is single-threaded. When applied to the program in Figure 18, the program produces consistent results. Within **run**, 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.

6.3 Handling Multiple Values

Racket supports multiple return values from functions, and these are used extensively in some libraries. We therefore explored three implementation strategies 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 [3] implement multiple values without explicit checks. Their idea hinges on 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 the control stack 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 Because JavaScript functions are objects, a limited kind of address arithmetic can be simulated to capture the

[7] 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 [4, 16], as done in Emscripten (`code.google.com/p/emscripten/`).

Figure 13 Measuring GOTO simulation on micro-benchmarks

Method	$N = 1000$	$N = 10000$	$N = 100000$	$N = 1000000$	$N = 10000000$
native	0.03 (0.17)	0.09 (0.29)	1.00 (0.00)	9.90 (0.30)	99.04 (1.37)
case/switch	0.01 (0.10)	0.14 (0.34)	1.41 (0.49)	14.40 (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)	345.10 (1.39)

N represents the number of calls. Time is in milliseconds (with standard deviation).
function/100 and function/1000 use trampoline ceilings of 100 and 10000 activation records respectively.

Figure 14 Comparing multiple-value return techniques

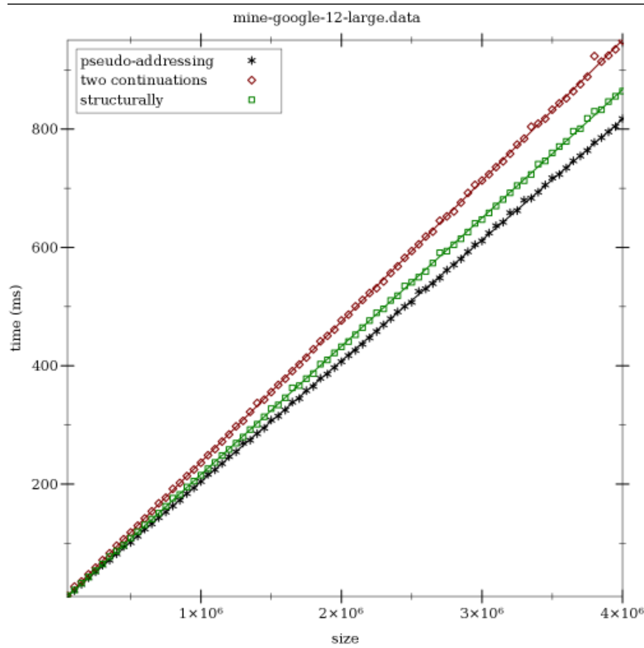
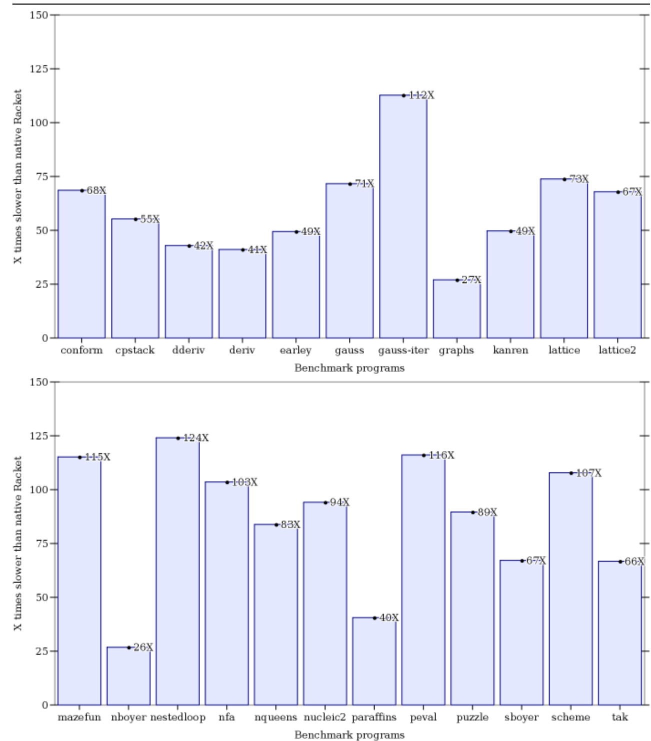


Figure 15 Whalesong vs. Racket's native JIT



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. This pseudo-addressing technique captures the essential features of Ashley and Dybvig's pointer arithmetic technique without increasing the traffic on the stack.

Figure 14 compares the relative performance of each technique on a pseudo-benchmark under Google Chrome 12 on an 3200Mhz AMD Phenom II X4 995. The pseudo-addressing technique demonstrates a lower constant overhead compared to the other two, and other browsers present similar results.

6.4 Wrapping Up

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 **run**'s maintenance of continuation marks allows program errors to be presented with source locations and stack traces relative to the user's original program.

Initial results show that the performance is within a factor of 50-100X of Racket. Given the great improvements in JavaScript evaluators, the fact that Whalesong programs are so much slower than ones in Racket might seem surprising. However, a Whalesong program's execution looks radically different from that of most JavaScript programs, due to the support for preemption and advanced control. These operations add significant execution overhead, and cause programs to look different from what JavaScript compilers are accustomed to optimizing.

There is also tremendous scope for improvement. A major factor is the overhead of the managed function call. The calling convention allows **run** 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 use a direct JavaScript call (as in the first prototype). It is future work to add this analysis to **compile**, so that it can eliminate, for simple functions, the overhead of **run**'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, because our focus has been on covering as much functionality as possible, and we have not yet had time to optimize it. (Also, the Whalesong numeric tower repeats the work of checking for overflow that JavaScript evaluators already perform to promote numbers from exact integers to floating-points.) In contrast, the Racket primitives have steadily improved over just shy of two decades, and many are implemented in C. Because Racket's primitives behave quite differently from those of JavaScript, we

do not benefit as much we would like from improvements in performance of JavaScript's primitives. Furthermore, our compiler currently does very little inlining, and because Racket programs make heavy use of primitives, most operations pay the full price of Whalesong's function applications.

7. Related Work

Many compilers now treat JavaScript as a target language. Emscripten (`code.google.com/p/emscripten/`) 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. However, these 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.

Canou, et al. [7] discuss compiling OCaml to JavaScript. They cover a similar design space, and arrive at similar conclusions. However, whereas our focus is on supporting the functional execution model, theirs is on supporting concurrency. Loitsch [14] 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.

Of contemporary systems, our I/O model is perhaps most closely related to that of Clean [2]. 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.

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

Our World model presents a similar scheme, with the pair of handlers:

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

The APIs for manipulating the graphical states are similar as well: both act on GUI elements that are uniquely identified, and both provide operations to change attributes of elements, such as adjusting the text or changing the visibility of windows. Our API's views have more of a tree-oriented flavor since the view is an abstraction over the browser DOM.

There are, however, 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 our model, the nonce value included in the view enables the runtime library to detect a similar kind of dependency. However, our model 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 is that our model 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 our model easier for beginners to use, since it does not requires knowledge of tuples or structured data, and both functions can be tested independently. 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 our model, 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.

The *make-world-event-handler* function plays a similar role to the *receiverE* mechanism used in Flapjax [15] 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 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 necessarily.

Figure 16 Program demonstrating a race-condition in JavaScript

```
// generate a random integer between [0, n)
var randInt = function(n) {
    return Math.floor(Math.random() * n);
};


// schedule a thunk f to be called at some
    point.
var schedule = function(f) {
    setTimeout(f, randInt(100));
};

var sharedValue = { n : 50 } ;

// decrement a value in sharedValue, and
    increment it again.
// intended to have no effect on the final
    value of sharedValue.n,
// but the use of schedule() allows certain
    unexpected control flow
// interleavings to occur:
var doWork = function() {
    var amount = Math.floor(randInt(sharedValue
        .n));
    sharedValue.n = sharedValue.n - amount;
    schedule(function() {
                 var newValue = sharedValue.n +
                      amount;
                 schedule(function() {
                     sharedValue.n = newValue;
                     });
             });
};

var afterLoad = function() {
    var i = 0;
    for (i = 0; i < 10; i++) {
        schedule(doWork);
    }

    // after waiting for the computation to
        complete...
    setTimeout(function() { alert(sharedValue.n
        ); },
               2000);
};
```

Figure 17 Implementing mutual exclusion 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 18 Revised version of `doWork` (figure 16) using mutexes

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

Acknowledgments This work was partially supported by the US NSF. We are grateful to Ethan Cecchetti, Matthew Flatt, Robby Findler, Kathi Fisler, Jay McCarthy, Emmanuel Schanzer, Jens Axel Søgaard, and Zhe Zhang. We thank the reviewers and Benjamin Lerner for constructive editing suggestions.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

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

[3] J. M. Ashley and R. K. Dybvig. An Efficient Implementation of Multiple Return Values in Scheme. In *Lisp and Functional Programming*, 1994.

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

[5] J. R. Bell. Threaded code. *Communications of the ACM*, 1973.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Implementation, Compilation, Optimization of Object Oriented Languages and Programming Systems*, 2009.

[7] B. Canou, E. Chailloux, and J. Vouillon. How to run your favorite language in web browsers. In *World Wide Web*, 2012.

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

[9] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System or, Fun for Freshman Kids. *International Conference on Functional Programming*, 2009.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, second edition, 2010. `www.ccs.neu.edu/home/matthias/HtDP2e/`.

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

[12] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. *International Conference on Functional Programming*, 2007.

[13] G. Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 1997.

[14] F. Loitsch. Exceptional Continuations in JavaScript. In *Scheme and Functional Programming*, 2007.

[15] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *Object-Oriented Programming Systems, Languages, and Applications*, 2009.

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

[17] E. Schanzer, K. Fisler, and S. Krishnamurthi. Bootstrap: Going beyond programming in after-school computer science. In *SPLASH Education Symposium*, 2013. `www.bootstrapworld.org`.

[18] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic Native Optimization of Interpreters. *Workshop on Interpreters, Virtual Machines, and Emulators*, 2003.

[19] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of Dynamic Languages using Hierarchical Layering of Virtual Machines. *Dynamic Languages Symposium*, 2009.

[20] D. Yoo, E. Schanzer, S. Krishnamurthi, and K. Fisler. WeScheme: The browser is your programming environment. In *Conference on Innovation and Technology in Computer Science Education*, 2011.