Putting in All the Stops: Execution Control for JavaScript

Samuel Baxter
University of Massachusetts Amherst
USA

Rachit Nigam
University of Massachusetts Amherst
USA

Joe Gibbs Politz
University of California San Diego
USA

Shriram Krishnamurthi
Brown University
USA

Arjun Guha
University of Massachusetts Amherst
USA

Abstract

Scores of compilers produce JavaScript, enabling programmers to use many languages on the Web, reuse existing code, and even use Web IDEs. Unfortunately, most compilers inherit the browser's compromised execution model, so long-running programs freeze the browser tab, infinite loops crash IDEs, and so on. The few compilers that avoid these problems suffer poor performance and are difficult to engineer.

This paper presents Stopify, a source-to-source compiler that extends JavaScript with debugging abstractions and blocking operations, and easily integrates with existing compilers. We apply Stopify to ten programming languages and develop a Web IDE that supports stopping, single-stepping, breakpointing, and long-running computations. For nine languages, Stopify requires no or trivial compiler changes. For eight, our IDE is the first that provides these features. Two of our subject languages have compilers with similar features. Stopify's performance is competitive with these compilers and it makes them dramatically simpler.

Stopify's abstractions rely on first-class continuations, which it provides by compiling JavaScript to JavaScript. We also identify sub-languages of JavaScript that compilers implicitly use, and exploit these to improve performance. Finally, Stopify needs to repeatedly interrupt and resume program execution. We use a sampling-based technique to estimate program speed that outperforms other systems.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords JavaScript, continuations, IDEs

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192370

ACM Reference Format:
Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. 2018. Putting in All the Stops: Execution Control for JavaScript. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192370

1 Programming On the Web

Scores of programming languages now compile to JavaScript and run on the Web [23] and there are several Web IDEs in widespread use [4, 6–8, 48, 57, 69, 74, 78, 83]. This growing audience for Web IDEs and languages that run in the browser includes both professionals and students. Unfortunately, JavaScript and the Web platform lack the abstractions necessary to build IDEs and serve as a complete compilation target for high-level languages. As a result, most compilers that produce JavaScript compromise on semantics and most Web IDEs compromise on basic debugging and usability features. Furthermore, as we explain in §7, new technologies such as WebAssembly and Web Workers do *not* address most of the problems that we address. Instead, our work may be viewed as presenting additional challenges to the creators of those technologies.

Limitations in Web IDEs The key problem facing a Web IDE is that JavaScript has a single-threaded execution environment. An IDE that wants to provide a "stop" button to halt runaway computation faces a nontrivial problem, because the callback for that button gets queued for execution behind the running code—which is not going to terminate. Related to this, to make pages responsive, the browser threatens to interrupt computations that run longer than a few seconds. This means long-running computations need to be broken up into short events. In turn, because computations cannot run for very long, JavaScript engines in browsers tend to provide very shallow stacks, which is problematic for functional programs that rely on recursion.

These limitations are not at all hypothetical. For example, Codecademy has a Web IDE for JavaScript and for Python. In response to several message board requests, Codecademy has a help article that explicitly addresses infinite loops that freeze the browser [81]: they suggest refreshing the page, which loses browser state and recent changes. Other systems, such as CodeSchool, Khan Academy, and Python Tutor [18], address this problem by killing all programs that run for more than a few seconds. These problems also afflict research-driven programming languages, such as Elm [11] and Lively Kernel [48], which crash when given an infinite loop. §7 discusses all these systems in more detail.

Some Web IDEs run user code on servers. However, this approach has its own limitations. The provider has to pay for potentially unbounded server time, servers must run untrusted code, and state (e.g., time) may reflect the server and not the client. Moreover, users have to trust servers, cannot work offline, and cannot leverage the browser's DOM environment. In this paper, we focus on IDEs that run user code in the browser.

Preliminary Solutions There are a handful of robust programming language implementations for the Web: GopherJS (Go) [17], Pyret [57], Skulpt (Python) [68], Doppio (JVM) [77], GambitJS (Scheme) [71], and Whalesong (Racket) [83]. They use sophisticated compilers and runtime systems to support some subset of long-running computations, shared-memory concurrency, blocking I/O, proper tail calls, debugging, and other features that are difficult to implement in the browser. However, these systems have several shortcomings.

First, these systems are difficult to build and maintain because they must effectively implement expressive control operators that JavaScript does not natively support. For example, GopherJS has had several issues in its implementation of goroutines [10, 12, 13, 15, 42, 72]; Skulpt [68] has had bugs in its debugger [14, 60]; and the Pyret IDE has problems (§6.4), several of which remain unresolved. In fact, the team that built Pyret previously developed a Web IDE for Scheme [84], but could not reuse the compiler from the Scheme system, because the execution control techniques and the language's semantics were too tightly coupled.

Second, these systems force all programs to pay for all features. For example, Pyret forces all programs to pay the cost of debugging instrumentation, even if they are running in production or at the command-line; GopherJS forces all programs to pay the cost of goroutines, even if they don't use concurrency; and Doppio forces all programs to pay the cost of threads, even if they are single-threaded. Third, these compilers have a single back-end—one that is presumably already complex enough—for all browsers, hence do not maximize performance on any particular browser. Finally, it is hard for a compiler author to try a new approach, since small conceptual changes can require the entire compiler and runtime system to change. Therefore, although these systems use techniques that are interchangeable in principle, in practice they cannot share code to benefit from each others' performance improvements and bug fixes. What is called for is a clear separation of concerns.

```
type Opts = {
  cont: 'checked' | 'exceptional' | 'eager', // continuation representation
  ctor: 'direct' | 'wrapped',                // constructor representation
  timer: 'exact' | 'countdown' | 'approx',   // time estimator
  yieldInterval: number,                     // yield interval
  stacks: 'normal' | 'deep',                 //  deep stacks
  implicits: true | false | '+',             // implicit conversions
  args: true | false | 'mixed' | 'varargs',  // arity-mismatch behavior
  getters: true | false,                     // support getters
  eval: true | false                         // apply stopify to eval'd code
}

type AsyncRun = {
  run: (onDone: () => void) => void,
  pause: (onPause: () => void) => void,
  resume: () => void
}

function stopify(source: string, opts: Opts): AsyncRun;
```

Figure 1. A portion of the `stopify` API.

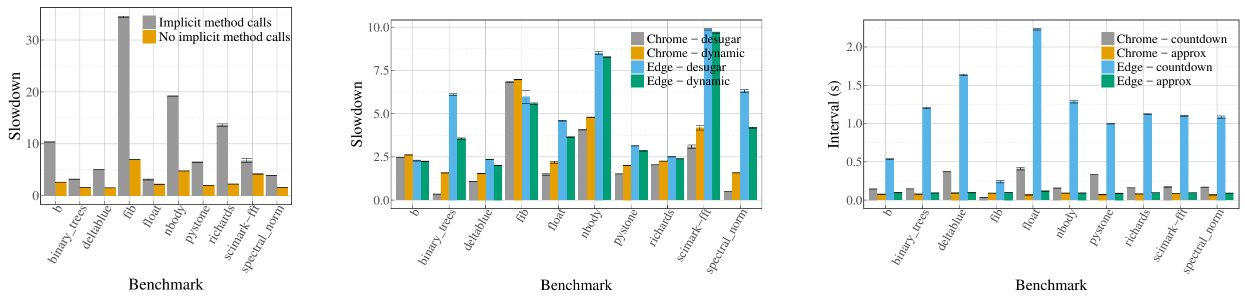
Our Approach Our goal is to enhance JavaScript to make it a suitable target for Web IDEs and, more broadly, to run a variety of languages atop JavaScript without compromising their semantics and programming styles. Our system, Stopify, is a *compiler from JavaScript to JavaScript*. Given a naïve compiler from language *L* to JavaScript—call it *L*JS—we can compose it with Stopify. Stopify prepares the code for Web execution while leaving *L*JS mostly or entirely unchanged.

Stopify relies on four key ideas. The first is to reify continuations with a family of implementation strategies (§3). The second is to identify reasonable sub-languages of JavaScript—as targeted by compilers—to reduce overhead and hence improve performance (§4). The third is to dynamically determine the rate at which to yield control to the browser, improving performance without hurting responsiveness (§5). Finally, we study how these different techniques vary in performance across browsers, enabling browser-specific performance improvements (§6).

Continuations and execution control features enable new capabilities for languages that compile to JavaScript. We show several: (1) Stopify supports long running computations by periodically yielding control to the browser; (2) Stopify provides abstractions that help compilers simulate blocking operations atop nonblocking APIs; (3) Stopify enables stepping debuggers via source maps; (4) Stopify allows simulating a much deeper stack than most browsers provide; and (5) Stopify can simulate tail calls on browsers that don't implement them natively.

We evaluate the effectiveness of Stopify in five ways:

1. We evaluate Stopify on ten compilers that produce JavaScript, nine of which require no changes, and quantify the cost of Stopify using 147 benchmarks (§6.1).
2. We use Stopify to build an IDE for nine languages. For eight of them, ours is the first Web IDE that supports long-running computation and graceful termination. For seven languages, our IDE also supports breakpoints and single-stepping (§5.2).



(a) Nonterminating arithmetic. **(b)** Constructor encoding. **(c)** Average time between yields.

Figure 2. Performance of Stopify relative to unmodified PyJS on a suite of 10 Python benchmarks run 10 times each. Each graph shows how an option setting in Stopify affects running time or latency. Error bars show the 95% confidence interval.

3. We apply Stopify to two well-known JavaScript benchmark suites and examine the difference in performance between them (§6.2).
4. We show that our Stopify-based Python IDE is faster and more reliable than a widely-used alternative (§6.3).
5. We present a case study of Pyret, integrating Stopify into its compiler and IDE. We show that Stopify makes the compiler significantly simpler, helps fix outstanding bugs, has competitive performance, and presents new opportunities for optimization (§6.4).

2 Overview

Stopify, which is written in TypeScript, provides a function called `stopify` (line 19 in Figure 1) that takes two arguments: (1) JavaScript code to run and (2) a set of options that affect the Stopify compiler and runtime system, which we elucidate in the rest of this section. The function produces an object with three methods:

1. The `run` method returns immediately and starts to evaluate the program. Stopify instruments every function and loop in the program to periodically save their continuation, yield control to the browser, and schedule resumption. In the absence of intervening events, the resumption event runs immediately (by applying the continuation). These periodic yields ensure that the browser tab remains responsive. When execution finally concludes, Stopify applies the callback passed to `run` to notify the caller.
2. The `pause` method interrupts the running program by setting a runtime flag that Stopify checks before applying a continuation in the resumption event. When the flag is set, Stopify calls the callback passed to `onPause` instead. Therefore, in an IDE, the event handler for a "stop" button can simply call `pause` and rely on Stopify to handle the rest.
3. Finally, `resume` resumes a paused program.

Stopify has additional methods to set breakpoints and simulate blocking operations, which we elide here. The rest of this section highlights a few features of Stopify using, as an example, the PyJS Python-to-JavaScript compiler's output as Stopify's input. This section presents only 10 benchmarks running on the Chrome and Edge Web browsers. §6 describes our experimental setup and an extensive evaluation.

Sub-languages of JavaScript There are some surprising ways to write nonterminating programs in JavaScript. For, instance, an apparent arithmetic expression like `x + 1` or field access like `o.f` can fail to terminate if the program has an infinite loop in a `toString` method or a getter, respectively. Stopify can stop these nonterminating programs, but at a significant performance cost. However, most compilers generate programs in a "well-behaved" sub-language of JavaScript. As Figure 2a shows, when we compile PyJS with conservative settings, the slowdown is much higher than when we specify that arithmetic expressions do not cause infinite loops, which is a safe assumption to make *of the code generated by PyJS*. §4 presents more sub-languages of this kind.

Browser-specific Optimizations Different browsers optimize JavaScript differently, which gives Stopify the opportunity to implement browser-specific optimizations. For example, any function may serve as a constructor and there are two ways to capture a constructor's continuation frame: we can (1) *desugar* constructor calls to ordinary function calls (using `Object.create`) or (2) *dynamically* check if a function is a constructor. Figure 2b plots the slowdown incurred by both approaches on Chrome and Edge. Desugaring is better than the dynamic approach on Chrome ($p = 0.001$), but the other way around in Edge ($p = 0.042$). To our knowledge, none of the works we cite in this paper implement browser-specific optimizations. §3 discusses compilation issues in more detail.

Estimating Elapsed Time Stopify needs to periodically interrupt execution and yield control to the browser's event loop, which can be done in several ways. Skulpt [68] continuously checks the system time, but this is needlessly expensive. Pyret [57] counts basic blocks and interrupts after executing a fixed number of them. However, Pyret's approach results in high variability in page responsiveness between benchmarks and browsers. Figure 2c shows the average time between interrupts when counting up to 1 million using this mechanism. On Chrome, a few benchmarks yield every 30 milliseconds, which is too frequent and needlessly slows programs down. In contrast, on Edge, some benchmarks only yield every 1.5 seconds, which is slow enough for the browser to display a *"script is not responding"* warning. In Stopify, we sample the system time and dynamically estimate the rate at which statements are executed. Using this mechanism, Stopify takes a desired interrupt interval, δ, as a parameter and ensures that the program interrupts every δ milliseconds with high probability. Figure 2c shows that the average time between interrupts with $\delta = 100$ is very close to δ on both browsers.

In sum, we've identified the JavaScript sub-language that PyJS targets, applied browser-specific optimizations, and used an estimator of system time. This combination makes Stopify's Python faster and more reliable than a widely-used alternative (§6.3), while also being simpler to maintain. Stopify has several more features and options that we use to support a variety of programming languages that the rest of this paper describes in detail.

3 Continuations for JavaScript

This section presents Stopify's first-class continuations for a fragment of JavaScript that excludes some of its most egregious features (which we defer to §4).

Language Extension We extend JavaScript with Sitaram and Felleisen's unary *control* operator [67], which we write as $\mathcal{C}$. When applied to a function, `C(function(k) { body })`, the operator reifies its continuation to a value, binds it to k, and then evaluates *body* in an empty continuation. Applying k aborts the current continuation and restores the saved continuation. For example, the result of the following expression is `0`:

```
10 + C(function (k) { return 0; })
```

Above, $\mathcal{C}$ evaluates the body in an empty continuation, thus produces `0`. In contrast, the next expression produces `11`:

```
10 + C(function (k) { return k(1) + 2; })
```

Above, $\mathcal{C}$ evaluates the body in an empty continuation, then the application in the body discards its continuation (which adds `2`) and restores the saved continuation (which adds `10`).

Stopify breaks up long running computations using a combination of $\mathcal{C}$ and browsers' timers. For example, the following function saves its continuation and uses `setTimeout` to schedule an event that restores the saved continuation:

```
function suspend() {
  C(function(k) { window.setTimeout(k, 0); }); }
```

Therefore, `suspend()` gives the browser a chance to process other queued events before resuming the computation. For example, the following infinite loop does not lock up the browser since each iteration occurs in a separate event:

```
var mode = 'normal';
var stack = [];

function P(f, g, x) {
  var t0, ℓ, k;
  if (mode === 'restore') {
    k = stack.pop();
    [t0] = k.locals;
    ℓ = k.label;
    k = stack[stack.length - 1];
  }
  function locals() { return [t0]; }
  function reenter() { return P(f,g,x); }
  if (mode === 'normal' || (mode === 'restore' && ℓ === 0)) {
    t0 = mode === 'normal' ? g(x) : k.reenter();
    if (mode === 'capture') {
      stack.push({ label: 0, locals: locals(), reenter: reenter });
      return;
    }
  }
  return f(t0);
}
```

Figure 3. An instrumented function.

```
while(true) { suspend(); }
```

We now discuss how we compile $\mathcal{C}$ to ordinary JavaScript.

Strawman Solutions There are two natural ways to implement $\mathcal{C}$ in modern JavaScript. One approach is to transform the program to continuation-passing style (CPS) (with trampolines if necessary). Alternatively, we could use generators to implement one-shot continuations [3]. We investigated and abandoned both approaches for two reasons. First, on several benchmarks, we find that CPS and generators are 3x and 2x slower than the approach we present below. Second, both approaches change the type of instrumented functions (CPS adds an extra argument and generators turn all functions into generator objects). This makes it hard to support features such as constructors and prototype inheritance.

3.1 Our Approach

We compile $\mathcal{C}$ to ordinary JavaScript in three steps: (1) A-normalize [16] programs, thus translating to a JavaScript subset where all applications are either in tail position or name their result; (2) box assignable variables that are captured by nested functions (discussed in §3.2.1); (3) instrument every function to operate in three modes: in *normal mode*, the program executes normally; in *capture mode* the program unwinds and reifies its stack; and in *restore mode* the program restores a saved stack. We run the program in the context of a driver loop that manages the transition between these modes.

We use the following function `P` as a running example:

```
function P(f, g, x) { return f(g(x)); }
```

Although `P` does not use $\mathcal{C}$ itself, the functions that it applies may use $\mathcal{C}$. Figure 3 shows an instrumented version of `P` and two global variables that determine the current execution mode (`mode`) and hold a reified stack (`stack`). In normal

$$\begin{aligned}
\mathcal{K}[\![x = e]\!] &= \textbf{if}\ (\mathit{normal})\ \{\ x{=}e\ \} \\
\mathcal{K}[\![s_1;\ s_2]\!] &= \mathcal{K}[\![s_1]\!];\ \mathcal{K}[\![s_2]\!] \\
\mathcal{K}[\![x = f_j(e_1 \cdots e_n)]\!] &= \textbf{if}\ (\mathit{normal}\ ||\ \ell === j)\ \{\ \mathcal{A}[\![x, f_j, e_1, \cdots, e_n]\!]\ \} \\
\mathcal{K}[\![\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2]\!] &= \textbf{if}\ ((\mathit{normal}\ \&\&\ e)\ ||\ (\mathit{restore}\ \&\&\ \ell \in s_1))\ \mathcal{K}[\![s_1]\!]\ \textbf{else if}\ (\mathit{normal}\ ||\ (\mathit{restore}\ \&\&\ \ell \in s_2))\ \mathcal{K}[\![s_2]\!] \\
\mathcal{K}[\![\textbf{function}\ f(x_1 \cdots x_n)\ \{\ \textbf{var}\ y_1 \cdots y_m;\ s\ \}]\!] &= \textbf{function}\ f(x_1 \cdots x_n)\ \{\ \textbf{var}\ y_1 \cdots y_m,\ \ell,\ k;\ \mathit{restoreFrame};\ \mathit{locals};\ \mathit{reenter};\ \mathcal{K}[\![s]\!]\ \} \\
\mathit{normal} &\triangleq \texttt{mode === 'normal'} \\
\mathit{restore} &\triangleq \texttt{mode === 'restore'} \\
\mathit{capture} &\triangleq \texttt{mode === 'capture'} \\
\mathit{restoreFrame} &\triangleq \textbf{if}\ (\mathit{restore})\ \{\ k = \texttt{stack.pop()};\ [y_1 \cdots y_m] = k\texttt{.locals()};\ k = k\texttt{.last};\ \} \\
\mathit{locals} &\triangleq \textbf{var}\ \texttt{locals = () => }[y_1 \cdots y_m] \\
\mathit{reenter} &\triangleq \textbf{var}\ \texttt{reenter = () => }f\texttt{.call(}\textbf{this}, x_1 \cdots x_n)
\end{aligned}$$

(a) Compiling JavaScript to support continuations.

$$\begin{aligned}
\mathcal{A}[\![x, f_j, e_1, \cdots, e_n]\!] = {}& x = \mathit{normal}\ ?\ f(e_1 \cdots e_n)\ :\ k\texttt{.reenter();} \\
& \textbf{if}\ (\mathit{capture})\ \{\ k\texttt{.push(\{ label: }j\texttt{, locals: locals(), reenter \});}\ \textbf{return};\ \} \\
C \triangleq {}& \textbf{function}\ \texttt{(f) \{ stack.push(\{ reenter: f \}); mode = 'capture' \}}
\end{aligned}$$

(b) Checked-return continuations.

$$\begin{aligned}
\mathcal{A}[\![x, f_j, e_1, \cdots, e_n]\!] = {}& \textbf{try}\ \{ x = \mathit{normal}\ ?\ f(e_1 \cdots e_n)\ :\ k\texttt{.reenter(); \}} \\
& \textbf{catch}\ \texttt{(exn) \{ }\textbf{if}\ (\mathit{capture})\ \{\ k\texttt{.push(\{ label: }j\texttt{, locals: locals(), reenter \});}\ \textbf{throw}\ \texttt{exn; \} \}} \\
C \triangleq {}& \textbf{function}\ \texttt{(f) \{ stack.push(\{ reenter: f \}); mode = 'capture';}\ \textbf{throw}\ \texttt{'capture'; \}}
\end{aligned}$$

(c) Exceptional continuations.

$$\begin{aligned}
\mathcal{A}[\![x, f_j, e_1, \cdots, e_n]\!] = {}& k\texttt{.push(\{ label: }j\texttt{, locals: locals(), reenter \});} \\
& x = \mathit{normal}\ ?\ f(e_1 \cdots e_n)\ :\ k\texttt{.reenter();} \\
C \triangleq {}& \textbf{function}\ \texttt{(f) \{ stack.push(\{ reenter: f \}); mode = 'capture';}\ \textbf{throw}\ \texttt{'capture'; \}}
\end{aligned}$$

(d) Eager continuations.

Figure 4. Compiling the C operator to ordinary JavaScript.

mode, the instrumented function is equivalent to the original function and the reified stack is not relevant.

Suppose `g(x)` applies `C`, which switches execution to capture mode. To address this case, `P` checks to see if the program is in capture mode after `g(x)` returns (line 16). If so, `P` reifies its stack frame (line 17) and returns immediately. A reified stack frame contains (1) a copy of the local variables, (2) a label that identifies the current position within the function, and (3) a thunk called `reenter` that re-enters the function.

Now, consider how `P` operates in restore mode. The function begins with a block of code that restores the saved local variables (lines 6—11). Next, on line 14, the function checks the saved label to see if `g(x)` should be applied. (The only label in this function is `0`.) Finally, instead of applying `g(x)` again, we apply the `reenter()` function that `g(x)` had pushed onto the stack during capture mode. When `g(x)` returns normally, the last line of the function calculates `f(t0)`, where `t0` is the result of `g(x)`.

In general, to implement the three execution modes, we transform a function `f` as follows. (1) We define a nested thunk called `locals` that produces an array containing the values of `f`'s local variables. (2) We define a nested thunk called `reenter` that applies `f` to its original arguments. (3) We give a unique label to every non-tail call in `f`. (4) After every non-tail call, we check if the program is in capture mode. If so, we push an object onto `stack` that contains the (a) label of the function call, (b) an array of local variable values (produced by `locals`), and (c) a reference to the `reenter` function. This object is the continuation frame for `f`. We then immediately return and allow `f`'s caller to do the same. (5) At the top of `f`, we add a block of code to check if the program is in restore mode, which indicates that a continuation is being applied. If so, we use the reified stack frame to restore local variables and the label saved on the stack. (6) We instrument `f` such that in restore mode, the function effectively jumps to the call site that captured the continuation. Here, we invoke the `reenter` method of the next stack frame. When the continuation is fully restored, execution switches back to normal mode. Finally, the top-level of the program needs to be wrapped by a thunk and executed within a driver loop that manages execution in two ways. (1) The expression `C(f)` switches execution to capture mode and reifies the stack. Therefore, the driver loop has to apply `f` to the reified stack. (2) When a continuation is applied, the program throws a special exception to unwind the current stack. The driver loop has to start restoring the saved stack by invoking the `reenter` method of the bottommost frame.

The function $\mathcal{K}$ in Figure 4a presents the compilation algorithm for a representative fragment of JavaScript. The algorithm assumes that expressions (e) do not contain function declarations or applications and that each non-tail function application is subscripted with a unique label. We write $\ell \in s$ to test if s contains an application subscripted with ℓ. We present the rule for capturing stack frames (the $\mathcal{A}$ function) and the definition of C in a separate figure (Figure 4b), because Stopify can capture continuations in other ways (§3.2).

3.1.1 Exception Handlers and Finalizers

Suppose a program captures a continuation within a **`catch`** clause. In restore mode, the only way to re-enter the **`catch`** clause is for the associated **`try`** block to throw an exception. Throwing an arbitrary exception will lose the original exception value, so we need to throw the same exception that was caught before. Therefore, we instrument each **`catch`** block to create a new local variable that stores the caught exception value and instrument each **`try`** block to throw the saved exception in restore mode.

Now, suppose a program captures a continuation within a **`finally`** clause. In restore mode, the only way to re-enter the **`finally`** block is to **`return`** within its associated **`try`** block. However, the returned value is not available within the **`finally`** block. Therefore, we instrument **`try`** blocks with finalizers to save their returned value in a local variable. In restore mode, the **`try`** block returns the saved value to transfer control to the **`finally`** block, which preserves the returned value.

3.2 Variations of Our Approach

The above approach is parameterizable in two major ways: (1) how stack frames are captures and (2) how continuations are captured within constructors.

Capturing Stack Frames Stopify can capture stack frames in three different ways. The previous section presented a new approach that we call *checked-return continuations*: every function application is instrumented to check whether the program is in capture mode. Stopify also supports two more approaches from the literature. An alternative that involves fewer checks is the *exceptional continuations* approach [36]. Here, $\mathcal{C}$ throws a special exception and every function application is guarded by an exception handler that catches the special exception, reifies the stack frame, and re-throws it. Although this approach involves fewer conditionals than the checked return approach, exception handlers come at a cost. Both checked-return and exceptional continuations reify the stack lazily. An alternative approach, which we call *eager continuations*, maintains a shadow stack at all times [71]. This makes capturing continuations trivial and very fast. However, maintaining the shadow stack slows down normal execution. A key feature of Stopify is that it unifies these three approaches and allows them to freely compose with all the other configuration options that it provides.

Constructors JavaScript allows almost any function to be used as a constructor and a single function may assume both roles. Stopify allows constructors to capture their continuation using two different approaches. The simple approach is to desugar **`new`**-expressions to ordinary function calls (using `Object.create`), which effectively eliminates constructors. (Constructors for builtin types, such as **`new`** `Date()`, cannot be eliminated in this way.) Unfortunately, this desugaring can perform poorly on some browsers.

It is more challenging to preserve **`new`**-expressions. Consider the following constructor, which creates two fields and calls a function `f`:

```
function N(x, f) {this.x = x; this.y = f(); return 0;}
```

We need to instrument `N` to address the case where `f` captures its continuation. The problem is that **`new`** `N(x, f)` allocates a new object (bound to **`this`**) every time it is called. Therefore, in restore mode, we have to apply `N` to the originally allocated object so that we do not lose the write to **`this`**`.x`. We address this problem in the `reenter` function, by calling `N` as a function (not a constructor) and passing the original value of **`this`**:

```
N.call(this, x, f)
```

This presents another problem—when `N` is invoked as a constructor (as it originally was) it returns **`this`**, but when `N` is applied as an ordinary function (during restoration), it returns `0`. Therefore, we also need to track if the function was originally invoked as a constructor, so that we can return the right value in restore mode. We accomplish this by using **`new`**`.target` (ES6) to distinguish function calls from constructor calls.

3.2.1 Assignable Variables

A problem that arises with this approach involves assignable variables that are captured by nested functions. To restore a function `f`, we have to reapply `f`, which allocates new local variables, and thus we restore local variables' values (e.g., line 8 in Figure 3). However, suppose `f` contains a nested function `g` that closes over a variable `x` that is local to `f`. If `x` is an assignable variable, we must ensure that after restoration `g` closes over the new copy of `x` too. We resolve this problem by boxing assignable variables that are captured by nested functions. This is the solution that *scheme2js* uses [36].

3.2.2 Proper Tail Calls

Our approach preserves proper tail calls. Notice that the application of `f` is in tail position and is not instrumented (line 21 of Figure 3). Consider what happens if `f` uses $\mathcal{C}$: `f` would first reify its own stack frame and then return immediately to `P`'s caller (instead of returning to `P`). In restore mode, `P`'s caller will jump to the label that called `P` and then call `nextFrame.reenter()`. Since `P` did not save its own stack frame, this method call would jump into `f`, thus preserving proper tail calls. On browsers that do not support proper tail calls, Stopify uses trampolines.

4 Sub-Languages of JavaScript

Stopify's continuations work with arbitrary JavaScript (ECMAScript 5) code, but this can come at a significant cost. Fortunately, compilers do not generate arbitrary code and every compiler we've studied only generates code in a restricted *sub-language of JavaScript*. This section identifies

Compiler	Impl	Args	Getters	Eval	(#) Benchmarks
PyJS	✗	M	✗	✗	(16) [9, 61]
ScalaJS	+	✗	✗	✗	(18) [9]
scheme2js	✗	V	✗	✗	(13) [30]
ClojureScript	+	M	✗	✗	(8) [9]
dart2js	+	✗	T	T	(15) [9, 73]
Emscripten	✗	V	✗	✗	(13) [9, 24]
BuckleScript	✗	✗	✗	✗	(15) [9, 47]
JSweet	+	M	✗	✗	(9) [9, 59]
JavaScript	✓	✓	✓	✓	(19) [9, 27]
Pyret	✗	✗	✗	T	(21) [57]

Figure 5. Compilers, their sub-language of JavaScript, and benchmark sources. A ✓ or ✗ indicates that a JavaScript feature is used in full or completely unused. The other symbols indicate restricted variants of the feature (discussed in §4).

sub-languages that compilers (implicitly) use. In fact, in several cases, we find that multiple, independently developed compilers use the same sub-language of JavaScript. Stopify makes these sub-languages explicit: the `stopify` function (Figure 1) consumes a program *p* and a specification of *p*'s sub-language and then exploits properties of the sub-language to produce simpler and faster code.

We classify each sub-language as a composition of four orthogonal JavaScript language features. Each feature is either completely unused (✗), used to its fullest extend (✓), or used in a restricted manner. Figure 5 summarizes the sub-languages that our ten compilers inhabit. Note that the only language that requires all JavaScript features is JavaScript itself! A compiler author can always use Stopify with all features enabled. But, targeting a sub-language of JavaScript will improve performance dramatically.

4.1 Implicit Operations

In JavaScript, an apparent arithmetic expression like `x - 1` may run forever. This occurs when `x` is bound to an object that defines a `valueOf` or `toString` method. The `-` operator implicitly invokes these methods when `x` is an object and these methods may not terminate. For completeness, Stopify supports all implicit operations, but they are expensive (Figure 2a). Fortunately, the only language that requires all implicits is JavaScript itself (the ✓ in the **Impl** column of Figure 5).

No Implicits Several compilers do not need JavaScript to make any implicit calls (✗ in the **Impl** column). For these compilers, Stopify can safely assume that all arithmetic expressions terminate.

Concatenation Only In JavaScript, the `+` operator is overloaded to perform addition and string concatenation, and some compilers rely on `+` to invoke `toString` to concatenate strings (`+` in the **Impl** column). For example, the JSweet Java compiler relies on this behavior, since Java overloads `+` and implicitly invokes `toString` in a similar way. For these compilers, Stopify desugars the `+` operator to expose implicit calls to `toString` and assumes that other operators do not invoke implicit methods.

4.2 Arity Mismatch Behavior

JavaScript has no arity-mismatch errors: any function may receive more arguments (or fewer) than it declares in its formal argument list. When a function receives fewer arguments, the elided arguments are set to the default value **`undefined`**. All the arguments, including extra ones, are available as properties of a special `arguments` object, which is an implicit, array-like object that every function receives. Some compilers do not leverage this behavior at all (✗ in the **Args** column). Stopify has full support for `arguments`, but we also identify two restricted variants that compilers use in practice.

Variable Arity Functions Many compilers use `arguments` to simulate variable-arity functions (**V** in the **Args** column). To restore the continuation frame of a variable-arity function, Stopify applies it to its `arguments` object instead of explicitly applying it to its formal arguments:

```
f.apply(this, arguments)
```

However, this simple approach breaks down when `arguments` is used in other ways.

Optional Arguments A different problem arises when `arguments` simulates optional arguments. Suppose a function `f` has formals `x` and `y` and that both are optional. If so, `f` can use `arguments.length` to check if it received fewer arguments than it declared and then initialize the missing arguments to default values. However, this does not affect `arguments.length`. Therefore, if we restore `f` by applying it to its `arguments` object (i.e., like a variable-arity function), then the default values will be lost. So, we need to restore `f` by explicitly applying it to its formal arguments, i.e., `f.call(this, x, y)`. This is how we restore ordinary functions (§3.1), thus we don't need a sub-language when the program uses optional arguments.

Mixing Optional and Variable Arity We've seen that variable-arity functions and functions with optional arguments need to be restored in two different ways. However, we also need to tackle functions that mix both features (**M** in the **Args** column). To restore these functions, we effectively apply both approaches simultaneously: we pass formal parameters explicitly and the `arguments` object as a field on the reified continuation frame.

Complete Support for Arguments However, even this does not cover the full range of possible behaviors. For example, JavaScript allows formal arguments to be aliased with fields in the `arguments` array. Therefore, if a function updates the same location as a formal argument and as an element of the `arguments` object then the approach above will break. Stopify supports this behavior by transforming all formal argument references into `arguments` object indexing. This comes at a higher cost to performance and is only necessary when the source language is JavaScript.

4.3 Getters, Setters, and Eval

In principle, any read or write to a field may invoke a getter or a setter that may have an infinite loop. Fortunately, this issue does not arise for most source languages (✗ in the **Getters** column). For example, Scala and Python support getters and setters, but ScalaJS and PyJS do not use JavaScript's getters and setters to implement them. On the other hand, Dart does make use of getters and setters but only calls trivial internal functions that terminate (**T** in the **Getters** column). Therefore, we can safely omit instrumentation for getters for Dart. If a compiler generates JavaScript with getters and setters, Stopify can instrument all field reads and writes to capture continuations. However, Stopify also supports a simple annotation (written as a JavaScript comment) that indicates that an expression may trigger a getter or a setter. Therefore, a compiler can be modified to produce this annotation where necessary, which avoids the cost of instrumenting every field access in the program, which can be prohibitively expensive.

Stopify supports `eval` by rewriting occurrences of `eval` to invoke the Stopify compiler, which is straightforward since it is written in TypeScript. However, the compiler and its dependencies are nearly 5MB of JavaScript and takes much longer than the browser's native `eval` function. Aside from interleaving the Stopify compiler with program execution, supporting `eval` requires heavier instrumentation that further degrades performance. For instance, all variables in scope at the time of `eval` must be conservatively boxed in case a variable escapes in the context of evaluating the string. Moreover, Stopify only supports strict `eval`; non-strict `eval` may introduce new variable bindings which can be referenced by the outer scope, and Stopify cannot distinguish between free variables and such cases. Fortunately, most languages do not require JavaScript's `eval` (✗ in the **Eval** column). (In fact, compilers tend not to support `eval` even when the source language requires it.) Dart and Pyret are two exceptions: they use `eval` as a form of compression: they dynamically generate lots of trivial functions (e.g., value constructors) that very obviously terminate (**T** in the **Eval** column). In these cases, it makes sense to leave `eval` uninstrumented. Finally, we note that the best way to support `eval` in the source language is to lightly modify the source language compiler to pass the `stopify` function an AST instead of a string. (Stopify uses a standard representation of JavaScript ASTs [1].) This avoids needlessly parsing and regenerating JavaScript at runtime.

5 Execution Control

We can now say more about the options that `stopify` (Figure 1) takes along with the program p to compile: (1) the implementation strategy for continuations (§3.1), (2) the sub-language that p inhabits (§4), (3) whether breakpoints and single-stepping are desired, and (4) whether p requires a deep stack. Stopify transforms p into an equivalent program that (1) runs without freezing the browser tab, (2) can be gracefully terminated at any time, (3) can simulate blocking operations, and (4) optionally supports deep stacks, breakpoints, and single-stepping.

```
var distance = 0, counter = 0, ticks = 0, lastTime = 0, velocity = 0;

function resetTime() { distance = 0; }

function estimateElapsed() {
  distance++;
  if (counter-- === 0) {
    const now = Date.now();
    velocity = ticks / (now - lastTime); lastTime = now; ticks = t * velocity;
    counter = ticks;
  }
  return distance / velocity;
}

// Stopify's implementation uses postMessage, which is faster than setTimeout.
function defer(k) { setTimeout(k, 0); } // enqueues k() in the event queue

var mustPause = false, saved;
function maySuspend() {
  if (estimateElapsed() >= δ) {
    resetTime();
    C(function (k) {
      return defer(function() {
        if (mustPause) { saved = k; onPause(); } else { k(); }
      }); }); } }
```

Figure 6. Stopify inserts `maySuspend()` into programs to support long computations.

Benchmark	Countdown ($\mu \pm \sigma$)	Approximate ($\mu \pm \sigma$)	Exact ($\mu \pm \sigma$)
b	122.4 ± 27.35 ms	87.75 ± 39.33 ms	114.3 ± 26.19 ms
binary-trees	146.3 ± 32.55 ms	89.18 ± 24.71 ms	108 ± 3.92 ms
deltablue	386.4 ± 85.99 ms	97.31 ± 21.06 ms	109.2 ± 1.359 ms
fib	67.63 ± 8.398 ms	98.44 ± 12.64 ms	109.3 ± 0.8464 ms
nbody	201.6 ± 39.57 ms	100 ± 29.88 ms	109.4 ± 0.8818 ms
pystone	315.1 ± 112.1 ms	89.1 ± 24.46 ms	109.5 ± 1.414 ms
raytrace-simple	237.2 ± 44.22 ms	98.84 ± 15.28 ms	109.4 ± 0.6615 ms
richards	214.2 ± 54.36 ms	95.57 ± 23.14 ms	109.5 ± 1.152 ms
spectral-norm	182.4 ± 62.91 ms	77.58 ± 27.61 ms	109.7 ± 1.249 ms

Figure 7. A comparison of the three time estimation strategies on a subset of Python benchmarks.

5.1 Long Computations and Graceful Termination

To support long-running computations, Stopify instruments p such that every function and loop calls the `maySuspend` function (Figure 6), which may interrupt the computation (by capturing its continuation with $\mathcal{C}$) and schedule it for resumption (using `defer`. The parameter δ determines how frequently these interrupts occur. These interruptions give the browser an opportunity to process other events, which may include, for example, a user's click on a "Pause" button. To support pausing, `maySuspend` checks if the `mustPause` flag is set and calls the `onPause` callback (from the `pause` method in Figure 1) instead of restoring the saved continuation.

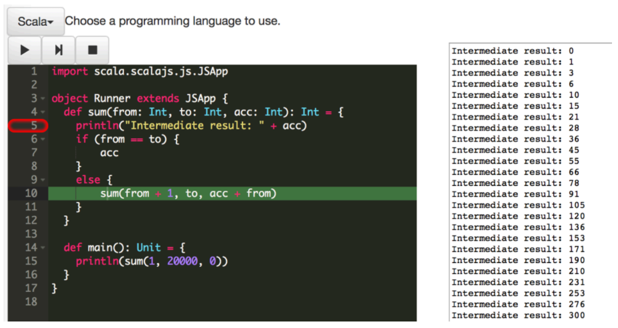
The `defer` function calls `estimateElapsed` to estimate how much time has elapsed since the last interruption. To do so, `estimateTime` counts the number of times it is called (`distance`) and maintains an estimate of the rate at which it is called (`velocity`). The parameter t determines how frequently the function actually checks the system time and thus the accuracy of the estimate. If the true function-call rate is v', then the estimated time will be off by a factor of $\frac{\texttt{velocity}}{v'}$, until we resample the system time.

Our approach is significantly less expensive that Skulpt's approach—which is to report the exact time elapsed—and more accurate than Pyret's approach—which is to assume a fixed execution rate for all programs. Since our approach relies on sampling, it does lose precision. The table in Figure 7 applies all three techniques to a subset of our benchmarks and reports the mean interrupt interval and its standard deviation for each case. In one particularly bad example, the mean interrupt interval (μ) is 108.3ms, with standard deviation (σ) 88.91ms. However, by Chebyshev's inequality—$\Pr(|X - \mu| \geq k\sigma) \leq 1/k^2$—even in this case, 95% of interrupt intervals will be less than 553ms, which is still low enough for browsers to be responsive.

5.2 Blocking, Deep Stacks, and Debugging

Stopify also allows programs to directly suspend and resume their execution with an API call. This allows the runtime system of a programming language to pause the program while it completes a nonblocking operation, like a network request or an input event, thereby simulating a blocking operation. In addition, certain browsers (e.g., Firefox and mobile browsers) have very shallow stacks, which is a problem for recursive programs. With the `'deep'` option for `stacks` (Figure 1), Stopify can simulate an arbitrary stack depth (up to heap size). This mode tracks a stack depth counter that is updated in every function call. On reaching a predefined limit, the stack is captured with $\mathcal{C}$, and computation resumes with an empty stack (that closes over the captured one). This counter needs just one variable, and so has negligible impact on performance for programs that don't trigger the stack depth counter: i.e., programs that don't need it hardly pay for it. This feature is important to languages like Pyret (§6.4), which encourages functional programming and abstracts low-level details such as the stack size.

Finally, Stopify can be configured to enable breakpoints and stepping. It does this by instrumenting the program to invoke `maySuspend` before every statement. For breakpoints, `maySuspend` checks if the currently-paused statement has a breakpoint set. For stepping, we treat the program as if breakpoints are set on every statement. Stopify exploits JavaScript *source maps* to allow breakpoints to be set using locations in the source program. Source maps are an established technology that is supported by seven of the compilers that we use in our evaluation (§6). For these languages, Stopify is the first Web IDE that supports breakpoints and single-stepping, and with no language-specific effort. (Figure 8 shows the Web IDE.)



```
import scala.scalajs.js.JSApp

object Runner extends JSApp {
  def sum(from: Int, to: Int, acc: Int): Int = {
    println("Intermediate result: " + acc)
    if (from == to) {
      acc
    }
    else {
      sum(from + 1, to, acc + from)
    }
  }

  def main(): Unit = {
    println(sum(1, 20000, 0))
  }
}
```

```
Intermediate result: 0
Intermediate result: 1
Intermediate result: 3
Intermediate result: 6
Intermediate result: 10
Intermediate result: 15
Intermediate result: 21
Intermediate result: 28
Intermediate result: 36
Intermediate result: 45
Intermediate result: 55
Intermediate result: 66
Intermediate result: 78
Intermediate result: 91
Intermediate result: 105
Intermediate result: 120
Intermediate result: 136
Intermediate result: 153
Intermediate result: 171
Intermediate result: 190
Intermediate result: 210
Intermediate result: 231
Intermediate result: 253
Intermediate result: 276
Intermediate result: 300
```

Figure 8. Debugging Scala with Stopify, single-stepped to the highlighted line. The red gutter marker is a breakpoint.

Browser	Platform
Google Chrome 60	Windows 10, Core i5-4690K, 16GB RAM
Mozilla Firefox 56	Windows 10, Core i5-4690K, 16GB RAM
Microsoft Edge 40	Windows 10, Core i5-4690K, 16GB RAM
Apple Safari 11	MacOS 10.13, Core i5-3470S, 8GB RAM
ChromeBook	ChromeOS, Celeron N3060, 4GB RAM

Figure 9. The platforms that we use to benchmark Stopify.

6 Evaluation

We now evaluate Stopify on a suite of ten languages. Stopify can introduce an order of magnitude slowdown or more, but its cost must be understood in context. Without Stopify, almost all benchmarks either freeze the browser tab or overflow the JavaScript stack. Furthermore, Stopify supports the first Web IDE with execution control for eight of these languages. The two exceptions are Python and Pyret: we directly compare Stopify to Web IDEs for these languages. This cost may be insignificant or irrelevant in a Web IDE, and once the code has been developed, Stopify can be dropped to deploy the full-speed version.

Platform selection We run all programs on the four major browsers (Chrome, Firefox, Edge, and Safari) and on a $200 ChromeBook. (See Figure 9 for more detailed specifications.)

6.1 Stopify on Nine Languages

For our first experiment, we report the cost of Stopify using the compilers and benchmarks in Figure 5. (We exclude Pyret here, devoting §6.4 to it.) For eight of these, we make *no changes to the compiler* and simply apply Stopify to the compiler output. The only exception is PyJS, which produces JavaScript embedded in a Web page: we modify the compiler to produce a standalone JavaScript file. When possible, we use the Computer Language Benchmarks Game [9], which is a collection of toy problems with solutions in several languages, as our benchmarks. We supplement the Shootout benchmarks with language-specific benchmarks in some cases. We only exclude benchmarks that use language or platform features that the compiler does not support (e.g.,

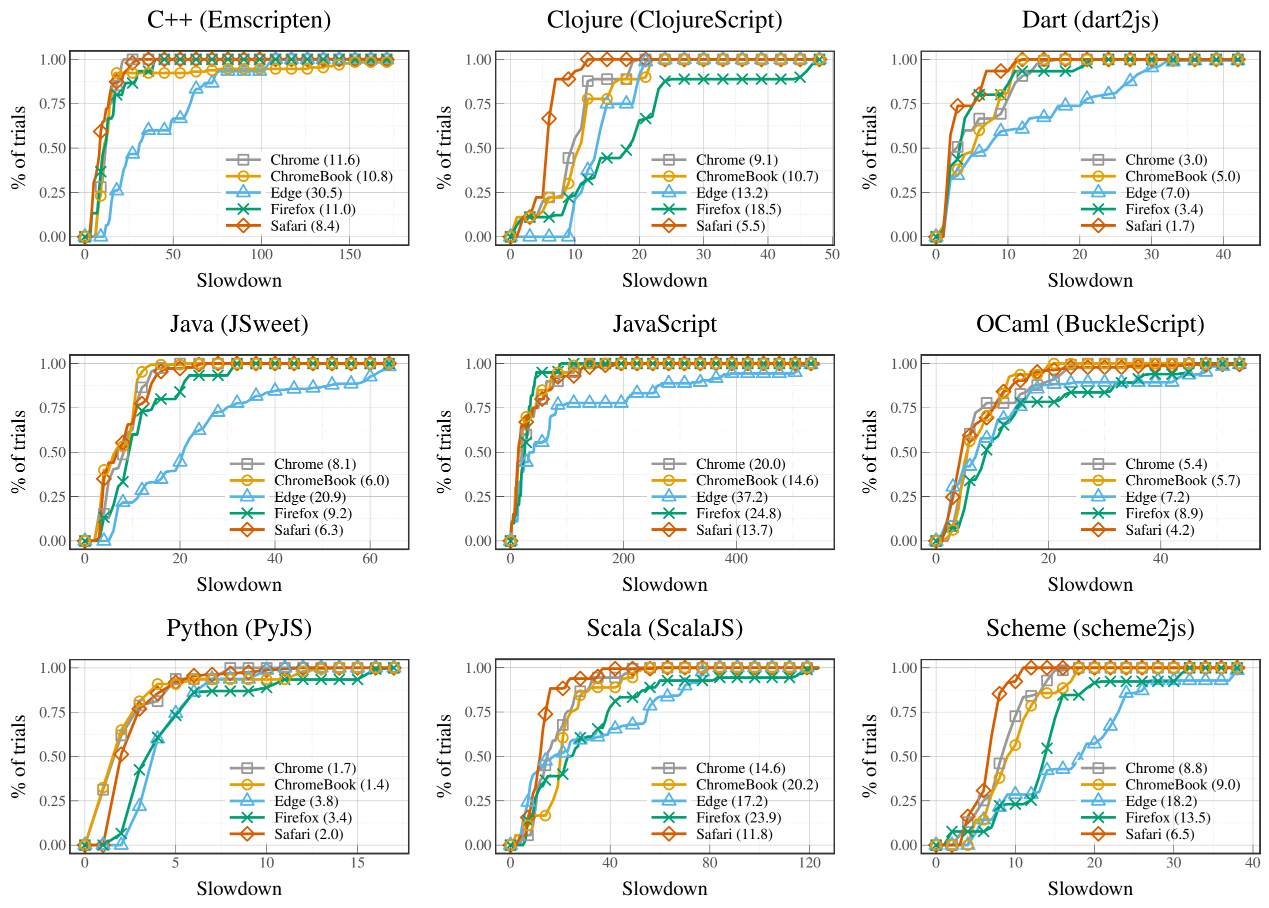


Figure 10. CDFs of Stopify's slowdown on nine languages. The median slowdown is in the legend.

Platform	Continuations	Constructors
Edge	checked-return	dynamic
Safari	exceptional	desugar
Firefox	exceptional	desugar
Chrome	exceptional	desugar

Figure 11. The best implementation strategy for continuations and constructors for each browser ($p < 0.01$).

files, foreign function interface, etc.). Therefore, the excluded benchmarks cannot run in the browser even without Stopify, so this is not a Stopify limitation. We run each benchmark ten times, both with and without Stopify. Finally, we ensure all benchmarks run for at least two seconds without Stopify by editing the number of iterations. This ensures that that the benchmark yields control several times and thus Stopify is properly exercised by every benchmark. We have 147 benchmarks across all ten languages.

For each compiler, we configure Stopify to exploit the sub-language it generates (Figure 5). Stopify also provides three strategies for implementing continuations and two strategies for supporting constructors (§3.1). We use microbenchmarks to choose the best settings for each browser (Figure 11). Finally, we configure Stopify to yield control every 100 ms, which ensures that the browser is very responsive. The slowdown that we report is the ratio of running time with and without Stopify.

We summarize this experiment in Figure 10, which shows empirical cumulative distribution functions (CDFs) of the slowdown for each language. In these graphs, the x-axis shows the slowdown and y-axis shows the fraction of trials with slowdown less than x. We also report the median slowdown for each platform in each graph's legend.

Discussion Figure 10 shows that (1) there is no best platform for Stopify, (2) the cost of Stopify depends on the source language compiler, and (3) the sub-language of JavaScript has a significant impact on performance.

We find that the slowdown tends to be much lower on Chrome and Safari than Edge and Firefox. However, we spent months developing Stopify using Chrome and Safari; thus, we speculate that the slowdowns on Firefox and Edge can be made much lower. We are pleasantly surprised that the slowdown on our ChromeBook is comparable to the slowdown on Chrome, despite the fact that the ChromeBook has far less cache and RAM than the desktop.

We also find that the cost of Stopify varies significantly by source language compiler. For example, the median slowdown on PyJS ranges from 1.7x—3.8x across all platforms. In contrast, Stopify performs more poorly on ScalaJS, with slowdowns ranging from 11.8x—23.9x. We attribute these differences to how these languages are compiled to JavaScript. ScalaJS directly translates the Scala standard library implementation to JavaScript, instead of mapping Scala data structures to JavaScript's builtin data structures. (ScalaJS exposes JavaScript data structures to Scala programs, but our benchmarks do not exploit them.) PyJS is more lightweight and maps Python data structures to JavaScript's builtin data structures. Since Stopify does not change function signatures, we could improve performance on ScalaJS by applying Stopify more selectively to the standard library.

Finally, the cost of Stopify crucially depends on identifying a sub-language of JavaScript that the compiler produces. The most extreme example is when the source language is JavaScript itself, so we cannot make any restrictive assumptions. Implicit operations and getters are the main culprit: the other nine languages either don't need them or use restricted variants of these features (Figure 5). We advise compiler writers who want to leverage Stopify to avoid these features. Fortunately, existing compilers already do. Still, JavaScript benchmarks that make less use of these features can avoid the substantial overhead seen in the worst cases. §6.2 examines JavaScript in more detail.

Effect on Code Size These benchmarks measure the end-to-end running time of JavaScript on a Web page, but do not capture page load time. Code size is a major contributer to delays in load time, especially on mobile platforms. Stopify increases code-size by a factor of 8x on average with a standard deviation of 5x.

Native Baselines The slowdown that we report is the only meaningful slowdown for platforms that cannot run native code (e.g., ChromeBooks) and languages that only compile to JavaScript (e.g., Pyret). For other cases, we report the slowdown incurred by compiling to JavaScript instead of running natively in Figure 15. These slowdowns are not caused by Stopify, but by running code in the browser.

6.2 Case Study: JavaScript

As seen in §6.1, Stopify exhibits its worst performance when the source-language is JavaScript itself. We now compare the performance of programs compiled with Stopify across two well-known JavaScript benchmark suites, Octane [46] and Kraken [27].

Figure 13 shows the slowdowns of each of these benchmark suites in Chrome 65 when compiled with Stopify. We exclude three benchmarks from the Octane benchmark suite due to 1) limitations of the Babylon parser, 2) the use of event-handlers (which is beyond the scope of this paper), and 3) the use of non-strict `eval` described in §4.3. We find that the cost of Stopify differs greatly between these benchmark suites—the median slowdown across Octane is 1.3x, compared to a median slowdown of 41.0x across Kraken. We attribute this performance difference to the frequency of implicit calls in arithmetic operations: Stopify desugars arithmetic to make implicit calls explicit, and Kraken calls these functions up to an order of magnitude more often than Octane. This experiment shows that despite heavy instrumentation, Stopify can achieve low overhead on real JavaScript applications. Stopify's performance characteristics are nuanced and largely dependent on the program being instrumented itself.

6.3 Case Study: Python (Skulpt)

In §2 and §6.1, we applied Stopify to PyJS to get execution control for Python in the browser. We now compare our approach to Skulpt, which is another Python to JavaScript compiler that has its own execution control. Skulpt is also widely used by online Python courses at Coursera [65, 66], by Rice University's introductory computer science courses [69], by online textbooks [19, 40, 41], and by several schools [62, 75, 76]. Skulpt can be configured to either timeout execution after a few seconds or to yield control at regular intervals, similar to Stopify. Unfortunately, Skulpt's implementation of continuations is a new feature that is quite unreliable and often fails to resume execution after yielding. Therefore, we perform the following experiment that puts Stopify at a disadvantage: we configure Skulpt to neither yield nor timeout and compare its performance to Stopify configured to yield every 100 ms. Figure 12 shows the normalized runtime of Stopify: a slowdown of 1 is the same as the running time of Skulpt; lower means Stopify is faster. Stopify is substantially faster or competitive with Skulpt on all benchmarks, despite its handicap in this experiment.

Though both PyJS and Skulpt are reasonably mature implementations of Python, they have their differences: they each seem to pass only portions of the CPython test suite, and each fail on some benchmarks. Indeed, Skulpt passes only 8 of our 16 benchmarks. Nevertheless, we believe that this experiment shows that Stopify is already fast enough to support Python Web IDEs, while increasing their reliability.

6.4 Case Study: Pyret

Pyret [57] is a mostly-functional programming language that runs entirely in the browser; it supports proper tail calls, blocking I/O, a REPL, and interactive animations; it allows users to gracefully terminate running programs; and it takes care to not freeze the browser tab. Despite five years of engineering and thousands of users, Pyret still suffers from issues that produce wrong results or freeze the browser tab [22, 28, 29, 31–33, 50–55].

Benchmark	μ	95% CI
anagram	0.25	± 0.01
binary-trees	0.27	± 0.01
fib	0.25	± 0.00
gcbench	0.08	± 0.01
nbody	0.25	± 0.00
pystone	0.37	± 0.01
schulze	1.25	± 0.08
spectral-norm	0.36	± 0.01

Figure 12. Slowdown relative to Skulpt. (Stopify is faster when $\mu < 1$.)

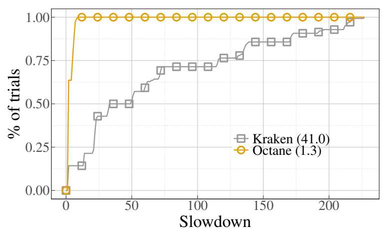


Figure 13. Slowdown on the Octane and Kraken benchmarks.

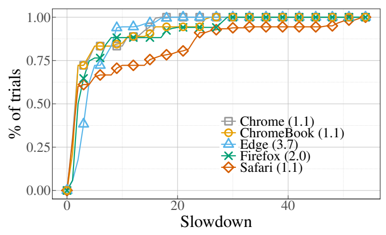


Figure 14. Slowdown with Pyret.

Compiler	Chrome			Firefox		
	μ	95% CI	Max.	μ	95% CI	Max.
Emscripten	2.59	± 0.42	9.98	3.72	± 0.73	16.95
ClojureScript	0.51	± 0.05	0.97	0.59	± 0.07	1.28
dart2js	1.32	± 0.10	2.69	3.20	± 0.62	13.29
JSweet	2.65	± 0.51	9.25	2.01	± 0.45	9.34
BuckleScript	20.48	± 11.80	475.89	67.98	± 28.96	1808.07
PyJS	6.87	± 1.19	27.17	4.40	± 0.59	12.26
ScalaJS	3.58	± 1.14	25.92	6.30	± 2.79	66.17
scheme2js	0.96	± 0.53	5.19	1.16	± 0.63	6.12

Figure 15. Slowdown incurred by compiling to JavaScript (without Stopify) relative to native.

```
function eachLoop(fun, start, stop) {
  var i = start;
  function restart(_) {
    var res = thisRuntime.nothing;
    if (--thisRuntime.GAS <= 0) { res = thisRuntime.makeCont(); }
    while(!thisRuntime.isContinuation(res)) {
      if (--thisRuntime.RUNGAS <= 0) { res = thisRuntime.makeCont(); }
      else {
        if(i >= stop) {
          ++thisRuntime.GAS;
          return thisRuntime.nothing;
        } else {
          res = fun.app(i);
          i = i + 1;
    } } }
    res.stack[thisRuntime.EXN_STACKHEIGHT++] =
      thisRuntime.makeActivationRecord("eachLoop", restart, true, [], []);
    return res;
  }
  return restart();
}
```

(a) Original, hand-instrumented implementation.

```
function eachLoop(fun, start, stop) {
  for (var i = start; i < stop; i++) { fun.app(i); }
  return thisRuntime.nothing;
}
```

(b) With Stopify, no instrumentation is necessary.

Figure 16. A higher-order function from Pyret's standard library that applies a function to a sequence of numbers.

Current Pyret Implementation The final phase of the Pyret compiler is a single pass that both (1) translates Pyret expressions to JavaScript and (2) produces JavaScript that is instrumented to save and restore the stack. The core Pyret runtime system, which is 6,000 lines of JavaScript, also serves two roles: it (1) implements Pyret's standard library and (2) and carefully cooperates with compiled Pyret code. Since Pyret encourages functional programming, its standard library has several higher-order functions, which are implemented in JavaScript to improve performance. These JavaScript functions are instrumented by hand, since they may appear on the stack when Pyret needs to pause a user-written function.

Due to this close coupling, the compiler and runtime system are difficult to maintain. For example, the Pyret runtime has a function called `eachLoop` that applies a functional argument to a range of numbers. However, most of its implementation is dedicated to saving and restoring the stack (Figure 16a). In fact, this function has been rewritten several times to fix bugs in its instrumentation [34, 35, 56].

Pyret with Stopify We applied Stopify to Pyret, which simplifies both the compiler and runtime system. However, we also find that Stopify (1) largely maintains Pyret's performance, (2) supports Pyret's Web IDE, including animations and the REPL, and (3) exposes new bugs in Pyret.

For the other languages, we use Stopify as a blunt instrument: we leave an existing compiler and runtime system unchanged and apply Stopify to the output JavaScript. We apply Stopify more carefully to Pyret. First, we strip out the portions of the compiler that manage the stack and use Stopify instead. This shrinks the last phase of the compiler from 2,100 LOC to 1,500 LOC (nearly 30% smaller). Second, 900 LOC of Pyret's runtime system involve stack management. We remove the stack management logic, which shrinks the code to 350 LOC (60% smaller), and we modify Pyret's build process to apply Stopify to this code. For example, Stopify lets us remove *all* the instrumentation from `eachLoop` (Figure 16). We envision that compiler authors who choose to use Stopify themselves will use it in this way to instrument only the necessary fragments of the language's libraries.

For the other nine languages, we use Stopify to only support long-running programs, stepping, and graceful termination. However, Pyret requires many more features (REPL, animations, blocking I/O). All these features are implemented in JavaScript and use an internal function to pause execution while they do their work. We made these features work by simply replacing Pyret's pause function with Stopify APIs.

Finally, Stopify exposed new and nontrivial bugs in Pyret's stack saving mechanism [45]. Two functions in the Pyret runtime system wrongly assumed that they would never appear on the stack during continuation capture. By simplifying the runtime system, Stopify made these errors evident.

Performance Figure 14 compares Pyret with Stopify to the old Pyret compiler. On Chrome and Safari, the median slowdown is 1.1x and a number of benchmarks are *faster* with Stopify than with Pyret' own implementation of continuations. The median slowdown on Firefox (2.0x) and Edge (3.7x) is disappointing but, as mentioned in §6.1, Stopify is not tuned for these browsers.

Unfortunately, we also find that some of our benchmarks have more significant slowdowns (up to 20x). All these benchmarks share the following attribute: they are deeply recursive programs that require more stack space than the browser provides. Stopify supports deep stacks (§5.2), but our implementation performs far more poorly than Pyret's. We expect to address this problem in the near future: Stopify should be able to output exactly the kind of code that Pyret does to implement deep stacks.

Future Work There are several more ways to improve our performance on Pyret. For example, Pyret performs several optimizations in its last phase that are entangled with its implementation of continuations. We omit these optimizations in our prototype compiler. Once ported to Stopify, they can be applied to other languages too. Furthermore, there are several ways to simplify Pyret's libraries now that we have Stopify. For example, many functions use expensive abstractions, such as callbacks and continuations, to avoid appearing on the stack during continuation capture. We could rewrite these functions using loops and apply Stopify, which may improve performance.

7 Related Work

Web Workers Web Workers [79] are essentially isolated processes for JavaScript and a Web IDE can use them to terminate a user's program [5]. However, unlike Stopify, Workers do not provide richer execution control (e.g., pausing or breakpointing) or deep stacks. Unlike Stopify, they also have a limited interface: they cannot directly access the DOM, and can communicate only through special shared datatypes [44] or by message passing.

WebAssembly WebAssembly [20] is a new low-level language in Web browsers. As of this writing, garbage collection, threads, tail calls, and host bindings (e.g., access to the DOM) are in the feature proposal stage [80]. Therefore, WebAssembly currently does not provide enough features to prototype a Stopify-like solution, nor are we aware of any multi-language demonstrations comparable to §6. Nevertheless, as it matures, Web IDEs may want to switch to it. For that to happen, WebAssembly will need to support the kind of execution control that Stopify provides for JavaScript.

Browser Developer Tools All modern Web browsers include a suite of developer tools, including a debugger for the JavaScript running in a page. These tools address the issue of pausing and resuming execution, and even utilize source maps to allow stepping through execution in a non-JavaScript source language, but they rely on developer tools being open at all times. Furthermore, unlike Stopify, browser developer tools do not enable long running computations, support arbitrarily deep stacks, or provide a programming model for synchronous operations atop nonblocking APIs.

Runtime Systems for the Web There are a handful of compilers that have features that overlap with Stopify, such as continuations [36, 71, 83], tail calls [36, 57, 71, 83], and graceful termination [57, 71, 83]. GopherJS [17] supports goroutines using mechanisms related to Stopify too. These compilers commingle language translation and working around the platform's limitations. Stopify frees compiler authors to focus on language translation. In addition, Stopify adds continuations to JavaScript itself and supports features such as exceptional control flow, constructors, ES6 tail calls, and more. Furthermore, Stopify supports a family of continuation implementation strategies and we show that the best strategy varies by browser.

Doppio [77] and Whalesong [83] implement bytecode interpreters in the browser that do not use the JavaScript stack. Therefore, they can suspend and resume execution. However, since these are bytecode interpreters for other platforms (JVM and Racket, respectively), existing compilers and libraries would have to change significantly to use them. Browsix [58] acts as an "operating system" for processes in Web Workers. Therefore, it inherits Web Workers' restrictions: workers cannot share JavaScript values and cannot interact with the Web page. It also does not provide deep stacks. Stopify allows code to run in the main browser thread, enabling access to the DOM and allowing execution control for IDEs.

Pivot [39] isolates untrusted JavaScript into an iframe and rewrites the program to use generators, which allows blocking I/O between the iframe and the outside world. Stopify is not an isolation framework and implements blocking without generators or CPS (§3).

Continuations on Uncooperative Platforms Many past projects have investigated implementing continuations in other platforms that do not natively support them, from C to .NET [2, 21, 36, 43, 49, 70, 82]. They use a variety of strategies ranging from CPS with trampolines, to C's `setjmp` and `longjmp` to effectively provide tail calls. These systems do not provide Stopify's other features (§1).

Quasar [63] rewrites JVM bytecode to support lightweight threads and actors, but is quite different from Stopify since

```
j = 0
while (j < 1000000000):
  j = j + 1
```

(a) CodeSchool (exception).

```
var i = 0;
while (i++ < 1000000000);
alert(i);
```

(b) CodePen (wrong result).

```
import Window
loop : a -> a
loop x = loop x
main = plainText (loop "")
```

(c) Elm Debugger (infinite loop).

Figure 17. Example programs that show the limitations of several Web IDEs.

JavaScript has no analogue to low-level JVM instructions that Quasar uses. Kotlin Coroutines [26] supports asynchronous functions in Kotlin using a transformation similar to CPS. In JavaScript, Stopify's transformation performs better than CPS and is more compatible with existing code (§3).

There are a handful of prior implementations of continuations for JavaScript [37, 38, 82]. Unwinder [37] and debug.js [38] use continuations to prototype an in-browser JavaScript debugger. However, these system support a much smaller fragment of JavaScript than Stopify. Moreover, they were not designed to support languages that compile to JavaScript, thus they do not exploit JavaScript sub-languages to improve performance. We show that deliberately targeting a sub-language improves performance significantly (§6).

Other Web IDEs Codecademy has a Web IDE for JavaScript that does not have a "stop" button and the only way to interrupt an infinite loop is to refresh the page and lose recent work. CodeSchool has a Web IDE for Python that imposes a hard timeout on all computations. example, it cannot run a Python program that counts up to only 1 billion (§7). Instead, it aborts with the message, *"TimeoutError: The executor has timed out."* The same problem affects Khan Academy's Web IDE for JavaScript [25], which terminates loops with more than five million iterations, printing *"a while loop is taking too long to run"*. Codepen is a Web IDE that users to collaboratively develop Web applications (i.e., HTML, CSS, and JavaScript). The CodePen IDE also terminates long-running loops, but continues running the program at the next statement after the loop, thus produces the wrong result (§7). Elm [11] has a time-traveling debugger that can step through a program's DOM events. However, infinite loops crash the Elm debugger (§7). Python Tutor [18] cannot handle long-running Python programs. If a program takes too long, it terminates with a message saying that it is not supported. The Lively debugger [64] supports breakpoints and watching variables using an interpreter for a subset of JavaScript that is written in JavaScript. In contrast, Stopify compiles JavaScript to JavaScript.

8 Conclusion

We have presented Stopify, a JavaScript-to-JavaScript compiler that enriches JavaScript with execution control. Stopify allows Web IDEs to work around the limitations of the JavaScript execution model. We present a new compilation strategy to support continuations, identify sub-languages of JavaScript to improve performance, and improve the responsiveness/performance tradeoff. We evaluate Stopify by showing that it smoothly composes with compilers for a diverse set of ten programming languages. Stopify is open source and available at www.stopify.org.

Acknowledgments

We thank Ben Lerner for his contributions to the design of multiple stack management techniques in Pyret, and for teaching us about them and their tradeoffs; Emery Berger, John Vilk, and Robert Powers for insightful discussions; and the anonymous reviewers and our shepherd Christian Wimmer for their feedback. This work is supported by the U.S. National Science Foundation under grants CCF-1717636.

References

[1] Babylon: A JavaScript parser used in Babel. https://github.com/babel/babel/tree/master/packages/babylon. Accessed Nov 10 2017.
[2] Henry G. Baker. 1995. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. *ACM SIGPLAN Notices* 30, 9 (Sept. 1995), 17–20.
[3] Carl Bruggeman, Oscar Waddell, and Kent R. Dybvig. 1996. Representing Control in the Presence of One-shot Continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[4] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nokolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[5] Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. 2017. Scaling up Functional Programming Education: Under the Hood of the OCaml MOOC. In *ACM International Conference on Functional Programming (ICFP)*.
[6] Codecademy. Learn JavaScript. https://www.codecademy.com/learn/learn-javascript. Accessed Nov 10 2017.
[7] CodeSchool. Learn JavaScript Online. https://www.codeschool.com/learn/javascript. Accessed Nov 10 2017.
[8] Codio. https://codio.com. Accessed Nov 10 2017.
[9] The Computer Language Benchmarks Game. https://benchmarksgame.alioth.debian.org. Accessed Nov 10 2017.
[10] Confusing behavior with exceptions inside goroutines. https://github.com/gopherjs/gopherjs/issues/225. Accessed Nov 10 2017.
[11] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[12] defer + go routine/channel + blocking = possible crash. https://github.com/gopherjs/gopherjs/issues/381. Accessed Nov 10 2017.
[13] Deferring runtime.Gosched() causes error. https://github.com/gopherjs/gopherjs/issues/426. Accessed Nov 10 2017.
[14] execLimit not triggering on large xrange loops. https://github.com/skulpt/skulpt/issues/711. Accessed Nov 10 2017.

[15] fix for goroutine blocking bookkeeping. https://github.com/gopherjs/gopherjs/issues/209. Accessed Nov 10 2017.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] GopherJS. https://github.com/gopherjs/gopherjs. Accessed Nov 10 2017.

[18] Philip J. Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[19] Mark Guzdial and Barbara Ericson. CS Principles: Big Ideas in Programming. http://interactivepython.org/runestone/static/StudentCSP/index.html. Accessed Nov 10 2017.

[20] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[21] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990. Representing Control in the Presence of First-class Continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[22] Wayne Iba. too much recursion in 2048 example code. https://github.com/brownplt/pyret-lang/issues/555. Accessed Nov 10 2017.

[23] List of Languages that Compile to JS. https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. Accessed Nov 10 2017.

[24] JetStream 1.1. http://browserbench.org/JetStream/. Accessed Nov 10 2017.

[25] Khan Academy: Computer programming. https://www.khanacademy.org/computing/computer-programming. Accessed Nov 10 2017.

[26] Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines.

[27] Kraken. https://wiki.mozilla.org/Kraken. Accessed Nov 10 2017.

[28] Shriram Krishnamurthi. impossible(?) to kill infinite loop w/ reactor. https://github.com/brownplt/pyret-lang/issues/839. Accessed Nov 10 2017.

[29] Shriram Krishnamurthi. calling plotting functions inside a reactor makes program unstoppable. https://github.com/brownplt/pyret-lang/issues/1089. Accessed Nov 10 2017.

[30] Larceny Benchmarks. http://www.larcenists.org/benchmarks2009all.html. Accessed Nov 10 2017.

[31] Benjamin S. Lerner. An infinite loop hangs big-bang. https://github.com/brownplt/pyret-lang/issues/508. Accessed Nov 10 2017.

[32] Benjamin S. Lerner. Responsiveness for rendering huge data at the REPL. https://github.com/brownplt/code.pyret.org/issues/37. Accessed Nov 10 2017.

[33] Benjamin S. Lerner. Bignums considered harmful. https://github.com/brownplt/pyret-lang/issues/1118. Accessed Nov 10 2017.

[34] Benjamin S. Lerner. eachLoop was simply broken: it did not restore the stack properly. https://github.com/brownplt/pyret-lang/commit/812d1c. Accessed Nov 10 2017.

[35] Benjamin S. Lerner. Fix broken eachLoop. https://github.com/brownplt/pyret-lang/commit/b7f9c9. Accessed Nov 10 2017.

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

[37] James Long. Unwinder. https://github.com/jlongster/unwinder. Accessed Nov 10 2017.

[38] Amjad Masad. debug.js. https://github.com/amasad/debugjs.com. Accessed Nov 10 2017.

[39] James Mickens. 2014. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. In *IEEE Security and Privacy (Oakland)*.

[40] Brad Miller and David Ranum. Problem Solving with Algorithms and Data Structures using Python. http://interactivepython.org/runestone/static/pythonds/index.html. Accessed Nov 10 2017.

[41] Brad Miller and David Ranum. How to Think Like a Computer Scientist: Interactive Edition. http://interactivepython.org/runestone/static/thinkcspy/index.html. Accessed Nov 10 2017.

[42] Miscompilation of functions w/ defer statements. https://github.com/gopherjs/gopherjs/issues/493. Accessed Nov 10 2017.

[43] Neil Mix and Dan Grisby. Narrative JavaScript. https://sourceforge.net/projects/narrativejs. Accessed Nov 10 2017.

[44] Mozilla, Inc. SharedArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_O$bjects/SharedArrayBuffer. Accessed Nov 10 2017.

[45] Rachit Nigam. Unsafe calls in runtime need to be wrapped with safeCall. https://github.com/brownplt/pyret-lang/issues/1251. Accessed Nov 10 2017.

[46] Octane. https://developers.google.com/octane. Accessed Nov 10 2017.

[47] OPerf Micro. https://www.typerex.org/operf-micro.html. Accessed Nov 10 2017.

[48] Krzysztof Palacz. 2008. The Lively Kernel Application Framework. In *International Conference on Scalable Vector Graphics*.

[49] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from generalized stack inspection. In *ACM International Conference on Functional Programming (ICFP)*.

[50] Joe Gibbs Politz. Make module loading stack safe. https://github.com/brownplt/pyret-lang/issues/145. Accessed Nov 10 2017.

[51] Joe Gibbs Politz. Printing large values is not stack-safe. https://github.com/brownplt/pyret-lang/issues/146. Accessed Nov 10 2017.

[52] Joe Gibbs Politz. Stopping (indefinitely) stopped programs. https://github.com/brownplt/pyret-lang/issues/163. Accessed Nov 10 2017.

[53] Joe Gibbs Politz. Too much recursion. https://github.com/brownplt/pyret-lang/issues/213. Accessed Nov 10 2017.

[54] Joe Gibbs Politz. Caja interferes with stack management/events during big-bang. https://github.com/brownplt/pyret-lang/issues/512. Accessed Nov 10 2017.

[55] Joe Gibbs Politz. Stack management fails on shallow, but long-lasting, recursion. https://github.com/brownplt/pyret-lang/issues/596. Accessed Nov 10 2017.

[56] Joe Gibbs Politz. more fixing of eachLoop. https://github.com/brownplt/pyret-lang/commit/844454. Accessed Nov 10 2017.

[57] Joe Gibbs Politz, Benjamin S. Lerner, and Shriram Krishnamurthi. Pyret. https://www.pyret.org/. Accessed Jul 5 2017.

[58] Bobby Powers, John Vilk, and Emery D. Berger. 2017. Browsix: Bridging the Gap Between Unix and the Browser. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[59] Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark2/. Accessed Nov 10 2017.

[60] Programs can incorrectly terminate with yieldLimit set to suspend at regular intervals. https://github.com/skulpt/skulpt/issues/723. Accessed Nov 10 2017.

[61] PyPy Benchmarks. https://bitbucket.org/pypy/benchmarks. Accessed Nov 10 2017.

[62] PythonRoom. https://pythonroom.com. Accessed Nov 10 2017.

[63] Quasar: Lightweight Threads and Actors for the JVM. http://blog.paralleluniverse.co/2014/02/06/fibers-threads-strands/. Accessed Nov 10 2017.

[64] Christopher Schuster. 2012. *Reification of Execution State in JavaScript*. Master's thesis. University of Potsdam Germany.

[65] Charles Severance. Programming for Everybody. https://www.coursera.org/learn/python. Accessed Nov 10 2017.

[66] Charles Severance. Python Data Structures. https://www.coursera.org/learn/python-data. Accessed Nov 10 2017.

[67] Dorai Sitaram and Matthias Felleisen. 1990. Control Delimiters and Their Hierarchies. *LISP and Symbolic Computation* 3, 1 (May 1990),

67–99.
[68] Skulpt. http://www.skulpt.org. Accessed Nov 10 2017.
[69] Terry Tang, Scott Rixner, and Joe Warren. 2014. An Environment for Learning Interactive Programming. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*.
[70] David Tarditi, Peter Lee, and Anurag Acharya. 1992. No Assembly Required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 2 (June 1992), 161–177.
[71] Eric Thivierge and Marc Feeley. 2012. Efficient Compilation of Tail Calls and Continuations to JavaScript. In *Workshop on Scheme and Functional Programming*.
[72] Tight loop in goroutine never yields. https://github.com/gopherjs/gopherjs/issues/698. Accessed Nov 10 2017.
[73] Ton80. https://github.com/dart-lang/ton80. Accessed Nov 10 2017.
[74] TreeHouse. Beginning JavaScript. https://teamtreehouse.com/tracks/beginning-javascript. Accessed Nov 10 2017.
[75] Trinket. https://trinket.io. Accessed Nov 10 2017.
[76] Tynker. https://www.tynker.com. Accessed Nov 10 2017.
[77] John Vilk and Emery D. Berger. 2014. Doppio: Breaking the Browser Language Barrier. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[78] Vocareum. https://www.vocareum.com. Accessed Nov 10 2017.
[79] W3C. Web Workers. https://www.w3.org/TR/workers/. Accessed Nov 10 2017.
[80] WebAssembly: Features to Add after the MVP. https://github.com/WebAssembly/design/blob/71c97d/FutureFeatures.md. Accessed Nov 10 2017.
[81] Why is my browser freezing when I submit an exercise? https://help.codecademy.com/hc/en-us/articles/220803187. Accessed Nov 10 2017.
[82] James Wright. JWACS. http://chumsley.org/jwacs/index.html. Accessed Nov 10 2017.
[83] Danny Yoo and Shriram Krishnamurthi. 2013. Whalesong: Running Racket in the Browser. In *Dynamic Languages Symposium (DLS)*.
[84] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2011. WeScheme: The Browser is Your Programming Environment. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.