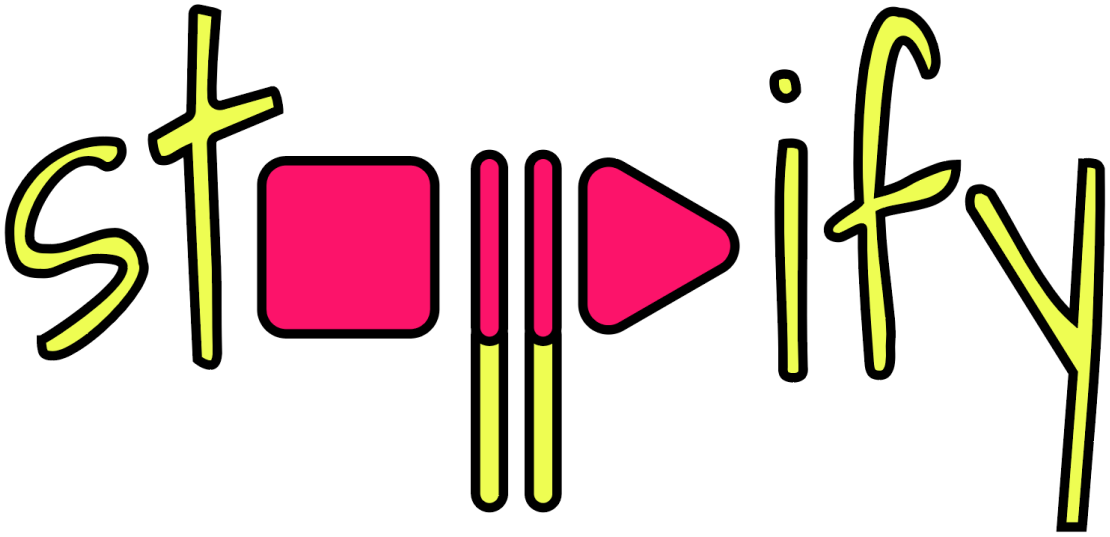
Stopify User's Manual

Release 0.6.0

Mar 15, 2019

Stopify User's Manual



by Samuel Baxter, Arjun Guha, Shriram Krishnamurthi, Rachit Nigam, and Joe Gibbs Politz

CHAPTER 1

Table of Contents

1.1 Quick Start

This section guides you through installing Stopify and applying it to a simple JavaScript program that runs forever and periodically prints the current time. However, since the program never yields control to the browser's event loop (e.g., using `setTimeout`), nothing will appear on the page and the browser tab will eventually crash. Stopify will make this program behave more naturally and actually show the output.

1.1.1 Instructions

1. Install the Stopify executable using NPM (or Yarn):

```
npm install -g stopify
```

2. Save the following program to the file `input.js`.

```
const elt = document.createElement("div");
document.body.appendChild(elt);

var i = 0;
var j = 0;
while (true) {
  if (i++ == 10000000) {
    j++;
    elt.innerText = "Still running ... " + (new Date());
    i = 0;
  }
}
```

This program will make any web browser crash.

3. Use the Stopify compiler to stopify the program:

```
stopify input.js output.js
```

4. The Stopify installation includes a copy of the Stopify runtime system (`stopify.bundle.js`). Look up the path using the following command:

```
stopify-path stopify.bundle.js
```

5. Create a simple web page that first loads the Stopify runtime system (i.e., the path produced in the previous step) and then uses the runtime system to load the saved file. An example is given below.

```
<html>
  <body>
    <script src="stopify.bundle.js"></script>
    <script>
    var runner = stopify.stopify("output.js");
    runner.console = console;
    runner.run(() => console.log("done"));
    </script>
  </body>
</html>
```

1.1.2 Testing

Finally, open the page in any browser. You'll find that the program "just works" and periodically prints the current time. In contrast, if you load `input.js` directly, it will not print anything and will eventually crash the browser tab.

1.2 Overview

Stopify is a JavaScript-to-JavaScript compiler that makes JavaScript a better target language for high-level languages and web-based programming tools. Stopify enhances JavaScript with debugging abstractions, blocking operations, and support for long-running computations.

Suppose you have a compiler *C* from language *L* to JavaScript. You can apply Stopify to the output of *C* and leave *C* almost entirely unchanged. Stopify will provide the following features:

- Stopify will support long-running *L* programs without freezing the browser tab. In particular, programs can access the DOM and are not limited to Web Workers.
- Stopify can pause or terminate an *L* program, even if it is an infinite loop.
- Stopify can set breakpoints or single-step through the *L* program, if *C* generates source maps.
- Stopify can simulate an arbitrarily deep stack and proper tail calls. This feature is necessary to run certain functional programs in the browser.
- Stopify can simulate blocking operations on the web.

To support these feature, Stopify has two major components:

- A *Compiler* that transforms ordinary JavaScript to stopified JavaScript, and
- A *Runtime System* that runs stopified JavaScript provides an API for execution control.

You can run the Stopify compiler in three ways:

1. **Hosted on a web page**: This is the easiest way to use Stopify. Moreover, when the compiler is hosted on a web page, your system will be able to compile users' program even when they are offline.

2. **As a command-line tool**: If your system already compiles code on a server (e.g., you run an *L*-to-JS compiler that does not run in the browser), then you may wish to run the compiler on the server.
3. **As a Node library**: If your server is written in Node, you may wish to use the compiler as a library. However, note that Stopify may take several seconds to compile large programs (i.e., programs with thousands of lines of JavaScript) and block connections to the Node server.

Both the compile and runtime have several options. Some of these options only affect performance, whereas other options affect the sub-language of JavaScript that the compiler targets (which in turn may affect performance).

1.3 Compiler

1.3.1 Web-based Compiler

To run the compiler from a web page, include the script *stopify-full.bundle.js*. Use the following command to get the path to this script on your local machine:

```
stopify-path stopify-full.bundle.js
```

This bundle exposes the following function:

```
stopify.stopifyLocally(url: string, copts?: CompileOpts, ropts?: RuntimeOpts):␣
↪AsyncRun
```

1.3.2 Command-line Compiler

The Stopify CLI compiler requires the name of the input file and the name of the output file. In addition, the compiler has several optional flags:

```
stopify [compile-opts] input.js output.js
```

To load a compiled file in the browser, the Stopify runtime provides the following function:

```
stopify.stopify(url: string, opts?: RuntimeOpts): AsyncRun
```

1.3.3 Compiler as a Node Library

To use the Stopify compiler a Node library, first import the Stopify library:

```
const stopify = require('stopify');
```

This library exposes the following function:

```
stopify.stopify(url: string, copts?: CompileOpts): string
```

To load a compiled file in the browser, the Stopify runtime provides the following function:

```
stopify.stopify(url: string, opts?: RuntimeOpts): AsyncRun
```

1.3.4 Compiler Options

The Stopify compiler accepts the following options:

```
interface CompilerOpts {
  captureMethod?: "lazy" | "catch" | "retval" | "eager" | "original", // --transform␣
↪from the CLI
  newMethod?: "wrapper" | "direct",                                   // --new from␣
↪the CLI
  getters?: boolean,                                                  // --getters␣
↪from the CLI
  debug?: boolean,                                                    // --debug from␣
↪the CLI
  eval?: boolean,                                                     // --eval from␣
↪the CLI
  eval2?: boolean,                                                     // --eval2␣
↪from the CLI
  es?: "sane" | "es5",                                                // --es from␣
↪the CLI
  hofs: "builtin" | "fill",                                           // --hofs from␣
↪the CLI
  jsArgs?: "simple" | "faithful" | "full",                            // --js-args␣
↪from the CLI
}
```

If an option is not set, Stopify picks a default value that is documented below. By default, Stopify is *not* completely faithful to the semantics of JavaScript (certain JavaScript features are difficult to support and incur a high runtime cost). Instead, Stopify's default values work with a number of compilers that we've tested. By default, Stopify does not support getters, setters, eval, builtin higher-order functions, implicit operations, `arguments`-object aliasing, and single-stepping. If you think you may need these features, you will need to set their corresponding flags.

Transformation (`.captureMethod`)

Stopify uses first-class continuations as a primitive to implement its execution control features. Stopify can represent continuations in several ways; the fastest approach depends on the application and the browser. The valid options are `"lazy"`, `"catch"`, `"retval"`, `"eager"`, and `"original"`. For most cases, we recommend using `"lazy"`.

Constructor Encoding (`.newMethod`)

Stopify implements two mechanisms to support suspending execution within the dynamic extent of a constructor call.

- `"wrapper"` desugars all `new` expressions to ordinary function calls, using `Object.create`.
- `"direct"` preserves `new` expressions, but instruments all functions to check if they are invoked as constructors, using `new.target`.

The fastest approach depends on the browser. We recommend using `wrapper`.

Eval Support (`.eval`)

How should Stopify handle JavaScript's `eval` function? By default, this flag is `false` and Stopify leaves `eval` unchanged. Since Stopify typically does not rename variables, using a stopfied program can use `eval`, but the evaluated code may lock-up the browser if it has an infinite loop.

If set to `true`, Stopify rewrites calls to JavaScript's `eval` function to invoke the Stopify compiler. (Note: Stopify does *not* rewrite `new Function` and dynamically generated `<script>` tags.) This allows Stopify to control execution

in dynamically generated code. Naturally, this requires the online compiler. However, the feature incurs considerable overhead.

Alternative Eval Support (`.eval2`)

The `eval2` flag implements an alternative approach to supporting JavaScript's `eval` function from within Stopified code. This flag is mutually exclusive with the `eval` compiler flag; only one of the two can be specified at compile-time.

If set to `true`, Stopify supports evaluating new code in the same global environment as the main program. This means that code executed by the `eval` function can refer to global variables and declare global variables that escape the scope of `eval`.

Implicit Operations (`.es`)

Stopify can suspend execution within user-written `valueOf()` and `toString()` methods that JavaScript invokes implicitly.

For example, the following program is an infinite loop in JavaScript:

```
var x = { toString: function() { while(true) { } } };
x + 1;
```

With the implicit operations flag is set to `"es5"`, Stopify will be able to gracefully suspend the program above. With the flag set to `"sane"`, Stopify will not be able to detect the the infinite loop. We have found that most source language compilers do not rely on implicit operations, thus it is usually safe to use `"sane"`.

Fidelity of `arguments` (`.jsArgs`)

The `arguments` object makes it difficult for Stopify to resume execution after suspension. Stopify supports `arguments` in full, but it also supports two simple special cases that improve performance.

- Use `"simple"` if the program (1) does not use `arguments` to access declared formal arguments and (2) only reads additional arguments using the `arguments` object.
- Use `"faithful"` if the program (1) does not use `arguments` to access declared formal arguments and (2) may read or write additional arguments using the `arguments` object.
- Use `"full"` for full support of JavaScript's `arguments` object.

Higher Order Functions (`.hofs`)

Programs cannot use bulitin higher-order functions (e.g., `.map`, `.filter`, etc.) with Stopify, since Stopify cannot instrument native code. The `.hofs` flag has two possible values:

- Use `"builtin"` if the program does not use any native higher-order functions.
- Use `"fill"` to have Stopify rewrite programs that use native higher-order functions to use polyfills written in JavaScript.

Getters and Setters (`.getters`)

Programs that suspend execution within getters/setters incur a lot of overhead with Stopify. The `.getters` flag has two possible values:

- Use `true` to have Stopify instrument the program to support suspension within getters and setters.
- Use `false` if the program does not use getters and setters.

Single-stepping and Breakpointing (`.debug`)

Set `.debug` to `true` to enable support for single-stepping and breakpointing. However, note that this requires more instrumentation and slows the program down further.

1.4 Runtime System

1.4.1 Runtime Configuration

The Stopify runtime system takes a dictionary of options with the following type:

```
interface RuntimeOpts {
  estimator?: "velocity" | "reservoir" | "exact" | "countdown" | "interrupt",
  yieldInterval?: number    /* must be greater than zero */,
  stackSize?: number        /* must be greater than zero */
  restoreFrames?: number    /* must be greater than zero */
}
```

The first two options control how frequently Stopify yields control to the browser (`yieldInterval`) and the mechanism that it uses to determine elapsed time (`estimator`). The last two options can be used to simulate a larger stack than what JavaScript natively provides.

Time estimator (`.estimator`)

By default, Stopify uses the `velocity` estimator that samples the current time (using `Date.now()`) and tries to yield every 100 milliseconds. The `velocity` estimator dynamically measures the achieved yield interval and adapts how frequently it yields accordingly. This mechanism is inexact, but performs well. You can adjust the yield interval, but we do not recommend using a value lower than 100.

The `reservoir` estimator samples the current time using *reservoir sampling* (i.e., the probability of resampling the current time decreases as the program runs longer). This technique is less robust than `velocity` to fluctuations in program behavior, but still outperforms other methods. This usually has a lower runtime overhead than `velocity`, but sacrifices accuracy. We recommend `velocity` for a more general, nondeterministic estimator.

The `countdown` estimator yields after exactly *n* yield points have passed. With this estimator, the `yieldInterval` is interpreted as the value of n and not a duration. We do not recommend using this estimator in practice, since a good value of n will depend on platform performance and program characteristics that are very hard to predict. However, it is useful for reproducing bugs in Stopify, since the `velocity` estimator is nondeterministic.

The `interrupt` estimator is a Node.js-only implementation which initializes a timer as a C++ extension to Node.js. With this estimator, a JavaScript `Buffer` object is signaled from C++ whenever a `yieldInterval` milliseconds has elapsed from the timer. This estimator is only supported in Node.js (it depends on native code), but experiments have shown that it implements the most precise estimation technique, with the smallest overhead in this environment.

Finally, the `exact` estimator checks the current time at every yield point, instead of sampling the time. This has a higher runtime overhead than `velocity` and we do not recommend it.

Unbounded stacks (`.stackSize` and `.restoreFrames`)

On certain browsers, the JavaScript stack is very shallow. This is a problem for programming languages that rely heavily on recursion (e.g., idiomatic functional code). If this is not a concern, you can ignore these options.

To support heavily recursion code, Stopify can spill stack frames on to the heap. Therefore, a program will *never* throw a stack overflow error (however, it may run out of memory). To do so, it tracks the depth of the JavaScript stack and spills stack frames when the stack depth exceeds `stackSize`. Similarly, when resuming computation, the `restoreFrames` parameter determines how many saved stack frames are turned into JavaScript stack frames.

To maximize performance, `stackSize` should be as high as possible and `restoreFrames` should be equal to `stackSize`. The largest possible value of `stackSize` depends on the source language and browser. In our experience, a value of 500 works well.

1.4.2 The `AsyncRun` Interface

```
interface NormalResult {
  type: 'normal';
  value: any;
}

interface ExceptionResult {
  type: 'exception';
  value: any;
  stack: string[]
};

type Result = NormalResult | ExceptionResult;

interface AsyncRun {
  run(onDone: (result: Result) => void,
      onYield?: () => void,
      onBreakpoint?: (line: number) => void): void;
  pause(onPaused: (line?: number) => void): void;
  resume(): void;
  setBreakpoints(line: number[]): void;
  step(onStep: (line: number) => void): void;
  pauseImmediate(callback: () => void): void;
  continueImmediate(result: Result): void;
  processEvent(body: () => any, receiver: (x: Result) => void): void;
}
```

The `AsyncRun` interface provides methods to run, stop, and control the execution of a stopified program. The interface provides several methods, none of which should be used directly by the stopified program. The following methods are meant to be used by the driver program that controls execution (e.g., a web-based IDE):

- The `run` method starts execution and requires a callback that gets invokes when execution completes. You may provide optional callbacks that are invoked when the program yields control and when a breakpoint is reached.
- The `setBreakpoint` method sets the active breakpoints.
- The `pause` method pauses the program at the next yield point and requires an optional callback that is invoked when the program has paused.
- The `resume` method resumes execution after a pause.
- The `step` method resumes execution and pauses again at the next yield point.

The following methods are are meant to be used by non-blocking JavaScript functions to provide simulated blocking interface to the stopified program:

- The `pauseImmediate` method suspends the stopified program and invokes the provided callback. A function should not execute anything after invoking `pauseImmediate`. Typically, a function that uses `pauseImmediate` will use it in a `return` statement.
- The `continueImmediate` function resumes execution with the provided value.

Illustrative Examples has several examples that use these methods to implement simulated blocking operations.

Finally, the `processEvent(f, onDone)` method allows external event-handlers to call a stopified function `f`. Since `f` may pause execution and thus not return immediately, Stopify passes its result to the `onDone` callback, which must not be a stopified function.

1.5 Illustrative Examples

This chapter presents several examples that showcase Stopify's features.

1.5.1 A Blocking `sleep` Function

The browser does not have a blocking `sleep` function. However, we can use `window.setTimeout` and Stopify to simulate a blocking `sleep` operation:

```
function sleep(duration) {
  asyncRun.pauseImmediate(() => {
    window.setTimeout(() => asyncRun.continueImmediate({ type: 'normal', value:␣
↪undefined }), duration);
  });
}
```

In the code above, `asyncRun` is an instance of `AsyncRun` (*The AsyncRun Interface*). Note that this function should be stopified itself and needs to be declared as an external. A complete example of a page that uses `sleep` is shown below.

```
<html>
  <body>
    <script src="../../stopify/dist/stopify-full.bundle.js"></script>
    <script>
    function sleep(duration) {
      asyncRun.pauseImmediate(() => {
        window.setTimeout(() => asyncRun.continueImmediate({ type: 'normal', value:␣
↪undefined }), duration);
      });
    }

    const program = `
      while(true) {
        sleep(1000);
        document.body.appendChild(document.createTextNode("."));
      }
    `;

    const asyncRun = stopify.stopifyLocally(program);
    asyncRun.g = { sleep, document, window, asyncRun };
```

(continues on next page)

(continued from previous page)

```
    asyncRun.run(() => { });
    </script>
  </body>
</html>
```

This program runs forever and prints a period each second.

1.5.2 A Blocking `prompt` Function

The `prompt` and `alert` functions that are built-in to browsers are not the ideal way to receive input from the user. First, modal dialog boxes are unattractive; second, a user can dismiss them; and finally, if a page displays too many modal dialog boxes, the browser can give the user to suppress all of them.

```
<html>
<body>
  <div id="webConsole"></div>
  <script src="../../stopify/dist/stopify-full.bundle.js"></script>
  <script>
    const webConsole = document.getElementById('webConsole');

    function alert(message) {
      const div = document.createElement('div');
      div.appendChild(document.createTextNode(message));
      webConsole.appendChild(div);
    }

    function prompt() {
      return runner.pauseImmediate(() => {
        const div = document.createElement('div');
        div.appendChild(document.createTextNode('> '));
        const input = document.createElement('input');
        div.appendChild(input);
        // When ENTER pressed, replace the <input> with plain text
        input.addEventListener('keypress', (event) => {
          if (event.keyCode === 13) {
            const value = input.value;
            div.appendChild(document.createTextNode(value));
            div.removeChild(input);
            runner.continueImmediate({ type: 'normal', value: value });
          }
        });
        webConsole.appendChild(div);
        input.focus();
      });
    }

    const program = `
      alert("Enter the first number");
      var x = Number(prompt());
      alert("Enter the second number");
      var y = Number(prompt());
      alert("Result is " + (x + y));
    `;
```

(continues on next page)

(continued from previous page)

```
    const runner = stopify.stopifyLocally(program);
    runner.g = { Number, prompt, alert, runner, document, webConsole };

    runner.run(() => console.log('program complete'));
  </script>
</body>
</html>
```

This program prompts the user for two inputs without modal dialog boxes.

1.5.3 External Events

```
<html>
<body>
  <button id="pauseResume">Pause / Resume</button>
  <div id="webConsole"></div>
  <script src="../../stopify/dist/stopify-full.bundle.js"></script>
  <script>
    const webConsole = document.getElementById('webConsole');

    var paused = false;
    document.getElementById('pauseResume').addEventListener('click', () => {
      if (paused) {
        paused = false;
        runner.resume();
      }
      else {
        runner.pause(() => {
          paused = true;
        });
      }
    });

    function alert(message) {
      const div = document.createElement('div');
      div.appendChild(document.createTextNode(message));
      webConsole.appendChild(div);
    }

    function onClick(callback) {
      window.addEventListener('click', evt => {
        runner.processEvent(
          () => callback(evt.clientX, evt.clientY),
          () => { /* result from callback does not matter */ });
      });
    }

    const program = `
      onClick(function(x, y) {
        alert('You clicked at (' + x + ', ' + y + ')');
      });
    `;

    const runner = stopify.stopifyLocally(program);
    runner.g = { onClick, alert, window, runner };
```

(continues on next page)

(continued from previous page)

```
    runner.run(() => console.log('program complete'));
  </script>
</body>
</html>
```

The `onClick` function in the code below is an example of an event handler that receives a stopified callback named `callback`. However, `onClick` cannot apply `callback` directly. To support suspending execution, `callback` must execute in the context of Stopify's runtime system.

Instead, `onClick` invokes Stopify's `processEvent` method, passing it a thunk that calls `callback`. If the program is paused, `processEvent` queues the event until the program is resumed. Therefore, if the user pauses the program, clicks several times, and then resumes, several clicks will clicks will occur immediately after resumption. It is straightforward to discard clicks that occur while paused by testing the `paused` variable before invoking `processEvent`.

1.6 Acknowledgements

Stopify draws inspiration from several existing systems, including debug.js, Doppio, Gambit.js, Pyret, scheme2js, Unwinder, and WeScheme. We thank Emery Berger, Benjamin Lerner, Robert Powers, and John Vilk for insightful discussions. We thank Caitlin Santone for designing the Stopify logo.

1.7 Release Notes

1.7.1 Stopify 0.6.0

- **Breaking change:** Global variables must now be initialized onto the `g` field of the Stopify `runner` object. This removes the `externals` API from the `AsyncRun` interface. See the examples for how to update code to this new interface.
- **Breaking change:** The `continueImmediate` function requires a `Result`. In previous releases, it would receive an ordinary value. This change allows external functions that pause stopified programs to resume with an exception.

  To upgrade old code, replace `continueImmediate(x)` with `continueImmediate({ type: 'normal', value: x })`.
- **Breaking change:** The `stopify-continuations` package now only provides the runtime components for continuation support. The Babel continuation compiler backend now exists in a new package, `stopify-continuations-compiler`.

  Code that only depends upon the `stopify` package directly does not need to be updated.
- Added the `catch` transform to the compiler. From experiments, `catch` often outperforms `lazy` and other transforms.
- Added the `eval2` compiler flag. This new mode supports evaluating new code in the same global environment as the main program.
- Estimator implementations have been migrated to a separate `stopify-estimators` package.
- Along with the `stopify-estimators` package, a new `interrupt` estimator is introduced for Node.js programs, using a C++ extension to Node.js to initialize a timer.

- Fixed several issues with breakpoints. See https://github.com/plasma-umass/Stopify/issues/424.

1.7.2 Stopify 0.5.0

- **Breaking change:** The `onDone` callback passed to `AsyncRun.run` always receives a `Result`. In previous releases, it would receive an optional error argument.
- Stopify now reports a stack trace when an an exception occurs in stopified code. However, stack traces only work with `captureMethod: lazy` (the default capture method).
- Setting the `debug` flag would crash the online compiler. This is now fixed.

1.7.3 Stopify 0.4.0

- Added an optional error argument to the `onDone` callback of `AsyncRun`. When the argument is present, it indicates that the stopified program threw an exception with the given error.

1.7.4 Stopify 0.3.0

- Cleanup and documented Stopify's Node API.
- Stopify can now execute blocking operations at the top-level. For example, the following program now works:

```
function sleep(duration) {
  asyncRun.pauseImmediate(() => {
    window.setTimeout(() => asyncRun.continueImmediate(undefined), duration);
  });
}

const asyncRun = stopify.stopifyLocally(`sleep(1000)`,
  { externals: [ 'sleep' ] }));
asyncRun.run(() => { });
```

  In previous versions of Stopify, this program would have raised an error.

  You could effective run this program by wrapping the blocking operation in a thunk, i.e., `function() { sleep(1000); }()`. However, this wrapping is now unnecessary.

- **Potentially breaking change:** Top-level variables declared within Stopify no longer leak into the global scope of the page. In previous versions of Stopify, top-level variables would leak as follows:

```
const asyncRun = stopify.stopifyLocally(`var x = 100;`);
asyncRun.run(() => {
  console.log(x); // prints 100
});
```

  This is no longer the case. However, this may break programs that relied on this behavior.

1.7.5 Stopify 0.2.1

- Fixed a bug introduced in release 0.2.0, where `stopifyLocally` would fail if run more than once.

1.7.6 Stopify 0.2.0

- Exposed the `velocity` estimator and set it as the default instead of `reservoir`. In our experiments, `velocity` performs better and has negligible overhead.
- Added the `.stackSize` and `.restoreFrames` runtime options, which allow Stopify to simulate an arbitrarily deep stack.
- Fixed a bug where programs that used `return` or `throw` in the `default:` case of a `switch` statement would not resume correctly.
- Added the `processEvent` function to the Stopify API.

1.7.7 Stopify 0.1.0

- Initial release

Stopify is a JavaScript-to-JavaScript compiler that makes JavaScript a better target language for high-level languages and web-based programming tools. Stopify enhances JavaScript with debugging abstractions, blocking operations, and support for long-running computations.

Suppose you have a compiler *C* from language *L* to JavaScript. You can apply Stopify to the output of *C* and leave *C* almost entirely unchanged. Stopify will provide the following features:

1. Stopify will support long-running *L* programs without freezing the browser tab. In particular, programs can access the DOM and are not limited to Web Workers.
2. Stopify can pause or terminate an *L* program, even if it is an infinite loop.
3. Stopify can set breakpoints or single-step through the *L* program, if *C* generates source maps.
4. Stopify can simulate an arbitrarily deep stack and proper tail calls. This feature is necessary to run certain functional programs in the browser.
5. Stopify can simulate blocking operations on the web.

In many cases, it is possible to "blindly" use Stopify by applying it to the output of your compiler. However, Stopify will compile faster, produce faster code, and support more features, if your compiler cooperates with Stopify in particular ways. This manual will guide you through using Stopify with your own compiler.

CHAPTER 2

Warning

This manual is a work in progress. Many Stopify features remain undocumented. We will preserve the interfaces documented here in subsequent releases of Stopify.