Implementing First-Class Continuations by Source to Source Translation

Arjun Guha

University of Massachusetts Amherst

Continuations Recap

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single “hole”, where the active expression used to be. We will write that hole as $\square$.

- Program: `console.log(2 + 3 * 4); console.log("hi")`
- Active expression: `3 * 4`
- Continuation `console.log(2 + □); console.log("hi")`

Note

All programming languages have continuations. They are fundamental for evaluation.

Capturing Continuations Recap

Languages with first-class continuations provide a primitive operator that can turn its own continuation into a value ("capturing the current continuation").

control is a unary operator, which takes a unary function as an argument. When applied to a function f, **control**(f):

1. Turns its own continuation into a continuation value k, and
2. Applies the function to the continuation value f(k) in the empty continuation.

Using first-class continuations, we can simulate a synchronous sleep function:

```
function sleep(n) {
  function sleeper(k) {
    setTimeout(k, n);
  }
  control(sleeper);
}

console.log("Three");
sleep(1000);
console.log("Two");
sleep(1000);
console.log("One");
sleep(1000);
console.log("Liftoff!");
```

Introduction

Today's topic: How to implement first-class continuations by source-to-source translation.

This material is based on the following paper:

Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in All the Stops: Execution Control for JavaScript. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018

Today's meta-topic: How to conduct research on a full-fledged programming language, without losing your sanity.

Several Languages Compile to JavaScript

1. JavaScript is nearly universal
2. Web-based demos are convenient
3. Compiling to JavaScript is easy

A few compilers for languages with interesting control operators:

	blocking I/O
Dart	generators
OCaml	concurrency

These compilers (and a few others) effectively implement first-class continuations in JavaScript.

Many other compilers drop support for control operators that JavaScript does not directly support.

There are several ways to implement first-class continuations. Which approach will work best for your programming language?

A very hard question to answer:

1. Cross-language performance evaluation is difficult: many other factors could affect performance
2. Changing the implementation of continuations is difficult: tends to affect many parts of the compiler and runtime system
3. The answer may be browser-dependent
4. The answer may be program-dependent (e.g., is continuation capture rare, or the norm?)

Stopify Overview

Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in All the Stops: Execution Control for JavaScript. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018

1. Stopify adds first-class continuations to JavaScript. In contrast, prior work supports first-class continuations in other languages that compile to JavaScript.[1]
2. Stopify implements some nice features atop continuations:
    1. Stopping and resuming the running program
    2. Running long computations without locking up the browser
    3. Breakpoints and single-stepping (at higher cost)
    4. Simulated synchronous I/O and support for web browser events
    5. Polyfills for native, higher-order functions that support continuation capture

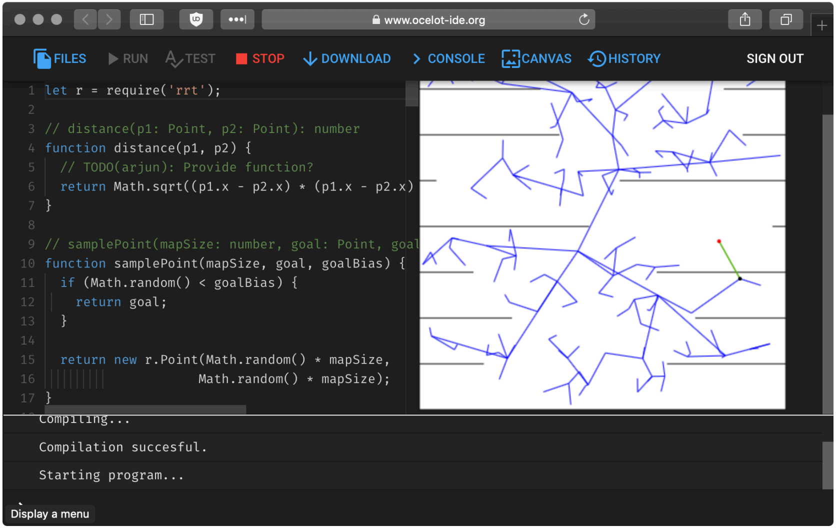
This is a common kind of pitch:

Stopify implements first-class continuations for JavaScript *once and for all*.

- Language-specific solutions are often faster in practice.
- Results suggest that Stopify is competitive or faster than language-specific approaches.
- Stopify uses a few tricks (cheats?) to achieve this result. We will see how later.

[1] There are a few earlier implementations, but they are significantly less complete than Stopify. See paper for details.

Stopify in the Classroom



```
let r = require('rrt');

// distance(p1: Point, p2: Point): number
function distance(p1, p2) {
  // TODO(arjun): Provide function?
  return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x)
}

// samplePoint(mapSize: number, goal: Point, goal
function samplePoint(mapSize, goal, goalBias) {
  if (Math.random() < goalBias) {
    return goal;
  }

  return new r.Point(Math.random() * mapSize,
                     Math.random() * mapSize);
}
```

```
Compiling...
Compilation succesful.
Starting program...
```

- Used by 200-300 students each semester in a sophomore software engineering class
- Goal is *not* to teach web programming
- Uses Stopify for synchronous I/O, long computations, stop button, etc.

Stopify Implementation: High-level idea

- The Stopify compiler instruments each function to run in three modes:
  - *Normal mode:* function runs normally
  - *Capture mode:* function saves its own stack frame and returns to its caller, which does the same
  - *Restore mode:* function restores its local variables and jumps to the last statement it was executing
- The Stopify runtime system manages the transitions between each mode
- The **`control`** operator sets the mode to capture
- Applying a continuation value clears the stack and sets the mode to restore

Example

```
 1 var saved = "nothing yet";
 2
 3 function save(k) {
 4   saved = k;
 5 }
 6
 7 function G(y) {
 8   console.log("Enter G");
 9   var z = control(save);
10   return y + z;
11 }
12
13 function F(x) {
14   console.log("Enter F");
15   var s = G(x + 10);
16   console.log(s);
17 }
18
19 F(0);
```

- How should we represent the continuation value captured at line 9?
- The continuation value must reconstruct the calls to `F` and `G`, without re-printing the output.

Continuation Representation

```
let saved = "nothing yet";

function save(k) {
  saved = k;
}

function G(y) {
  L1: console.log("Enter G");
  L2: var z = control(save);
  L3: return y + z;
}

function F(x) {
  L4: console.log("Enter F");
  L5: var s = G(x + 10);
  L6: console.log(s);
}

F(0);
```

```
let saved = [
    { f: F, l: L5, vars: { x: 0 } },
    { f: G, l: L2, vars: { y: 10 } }
];
```

- In capture mode, each function pushes its own frame into the array
- In restore mode, each function stores its own stack frame and then applies the next function in `kValue`

Restoring Continuations

Let's pretend we have the **goto** control operator in JavaScript:

```
function G(y) {
  if (mode === 'restore') {
    y = cont[0].vars.y;
    var l = cont[0].l;
    cont.shift();
    goto l;
  }
  L1: console.log("Enter G");
  L2: let z = control(save);
  L3: return y + z;
}

function F(x) {
  if (mode === 'restore') {
    x = cont[0].vars.x;
    var l = cont[0].l;
    cont.shift();
    goto l;
  }
  L4: console.log("Enter F");
  L5: let s = G(x + 10);
  L6: console.log(s);
}

F(0);
```

```
let saved = [
    { f: F, l: L5, vars: { x: 0 } },
    { f: G, l: L2, vars: { y: 10 } }
];
```

- `isRestoring` and `cont` are variables set by the Stopify runtime system
- The **control** operator is implemented as a function

```
function control(receiver) {
    if (mode === 'restore') {
        mode = 'normal';
        return /* continuation arg */;
    }
    else if (mode === 'normal') {
       // Start capturing
    }
    else {
        // impossible case
    }
}
```

Capturing Continuations

- To capture a continuation, the program has to save its stack
- The **control** operator throws an exception that contains an array
- Each function will catch the exception and and itself to the array

```
function G(y) {
  if (mode === ’restore’) {
    ...
  }
  L1: console.log("Enter G");
  try {
    L2: var z = control(save);
  }
  catch (exn) {
    exn.stack.push({
      f: G, l: L2,
      vars: { y: y }
    });
    throw exn;
  }
  L3: return y + z;
}

function F(x) {
    if (mode === ’restore’) {
      ...
    }
    L4: console.log("Enter F");
    try {
      L5: var s = G(x + 10);
    }
    catch (exn) {
      exn.stack.push({
        f: F, l: L5,
        vars: { x: x }});
```

```
let saved = [
    { f: F, l: L5, vars: { x: 0 } },
    { f: G, l: L2, vars: { y: 10 } }
];
```

```
function control(receiver) {
    if (mode === ’restore’) {
        ...
    }
    else if (mode === ’normal’) {
        throw {
          stack: [],
          receiver: receiver
        };
    }
    else {
        // impossible case
    }
}
```

Capture/Restore with Arbitrary Expressions

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

Same approach:

```

function P(f, g, x) {
  if (mode === 'restore') {
    ...
  }
  try {
    L1: return(f(g(x)));
  }
  catch (exn) {
    exn.stack.push(...);
    throw exn;
  }
}
```

Two serious issues:

- During capture: we cannot (easily) determine if the exception was thrown by `f` or `g`.
- During restore: **goto** `L1` always applies `g` first, and there is no way to skip it and apply `f`

(Automatically) rewrite `P` to the following equivalent function:

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

A Normal Form

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. PLDI 1993.

My other experiences with ANF:

- Type system design
- Program verification
- Program analysis

After A Normalization

P after A Normalization:

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

Correctly instrumented version of P:

```
function P(f, g, x) {
  if (mode === 'restore') {
    f = cont[0].vars.f;
    g = cont[0].vars.g;
    x = cont[0].vars.x;
    let l = cont[0].l;
    goto l;
  }
  L1: let t;
  try {
    L2: t = g(x);
  }
  catch (exn) {
    exn.stack.push({ f: P, l: L2,
      vars: { f: f, g: g, x: x } });
    throw exn;
  }
  L3: return f(t);
}
```

- Note that `P` does not capture/restore the application `f(t)`
- Therefore, when capture starts within `f`, the next frame saved in `cont` will be frame of the caller of `P`
- During restore, the caller of `P` will restore the `f`-frame (skipping the `P`-frame)
- This is safe, because the `P`-frame is "useless": it is merely: **return** `f(t)`
- This lets you fudge *proper tail calls*

JavaScript A Normalization Pipeline

Doing this correctly was half the work.
Multi-step process:

1. Desugar a few JavaScript features (e.g., `var x, y;`)
2. Add around loop/if bodies
3. Eliminate all loops except **`while`** loops
4. Eliminate **`continue`**
5. Eliminate **`switch`**
6. Eliminate short-circuiting boolean expressions (e.g., `f() || g()`)
7. A Normalize

Approach:

1. Straightforward assumptions and guarantees between each step
2. Some steps are not fundamental, but simplify later stages
3. Guiding principle: grammar of ANF-JavaScript
4. Only one optimization: do not A-normalize expressions without applications (e.g., `x + 1`[2])
5. Reusable artifact

[2] More later

Implementing Continuations by Source to Source Translation

1. A Normalize JavaScript (7+ steps)
2. Box assignable variables that are captured by nested functions and may be captured in continuations
   Analogous problem in compilers:

```
function F(x) {
  function g(y) {
    return x + y;
  }
  return g;
}
let addTen = F(10);
addTen(5); // is x on the stack?
```

3. Allow **goto** to function applications
4. Add capture/restore code to each function
5. One optimization: avoid capturing code when it is obvious that the called function will not capture its continuation ($\approx$ 30% performance improvement)

JavaScript

Everything discussed so far is JavaScript-neutral, and is applicable to other languages.

Arity Mismatch Errors

- The *arity* of a function is the number of arguments in takes.
- An *arity mismatch error* occurs when a function receives the wrong number of arguments. In statically typed languages (e.g., Java), arity mismatch errors occur before the program is run. In dynamically typed languages, arity mismatch errors occur while the program is running.
- JavaScript does not have arity mismatch errors.

```
function F(x) {
  console.log(x);
}

F(1, 2); // prints 1. The 2 is ignored.
F(); // prints undefined.
```

- This is not a problem for continuations

The “arguments” Object

All the arguments of a function are available in the `arguments` object.

```
function F(x) {
  console.log(x);
  for (let i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}

F(1,2,3,4); // prints 1 1 2 3 4
```

Problem

This is a problem: we may lose the extra arguments during capture.

Implicit Method Calls

Trivia

Give value for x, such that "Hello " + x is an infinite loop.
Hint: The same idea holds in Java too.

```
let x = {
  toString() {
    while (true) { }
  }
}
```

Problem

```
let x = {
  toString() {
      control(...)
  }
}
```

Solutions and Performance

- Problem: functions with arity mismatches
- Solution: save and restore the `arguments` object[3]
- Performance problem: `arguments` object materialization
- Problem: implicit method calls,
- Solution: make the implicit calls explicit

```
let t1 = x.toString(); // and valueOf
let t2 = "Hello " + t1;
```

- Performance problem: lots of extra instrumentation
- Problem: JavaScript supports getters and setters:

```
let a = o.b; // may call the getter b
```

- Solution: make the implicit calls explicit
- Performance problem: lots of extra instrumentation

[3] This does not work in the general case.

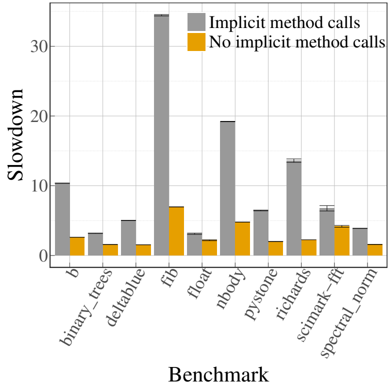
Solution: Ignore Problems When Possible

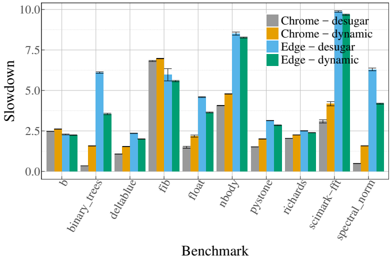
A compiler that produces JavaScript emits code into a sub-language of JavaScript.

Compiler	**Impl**	**Args**	**Getters**	**Eval**	**(#) Benchmarks**
PyJS	✗	**M**	✗	✗	(16) PyPy Benchmarks, Shootout
ScalaJS	+	✗	✗	✗	(18) Shootout
scheme2js	✗	**V**	✗	✗	(13) Larceny
ClojureScript	+	**M**	✗	✗	(8) Shootout
dart2js	+	✗	**T**	**T**	(15) Ton80, Shootout
Emscripten	✗	**V**	✗	✗	(13) JetStream, Shootout
BuckleScript	✗	✗	✗	✗	(15) OPerf-Micro, Shootout
JSweet	+	**M**	✗	✗	(9) SciMark, Shootout
JavaScript	✓	✓	✓	✓	(19) Kraken, Shootout
Pyret	✗	✗	✗	**T**	(21) Pyret

- A ✓ or ✗ indicates that a JavaScript feature is used in full or completely unused. The other symbols indicate restricted variants of the feature.
- Identifying the right sub-language can improve performance dramatically

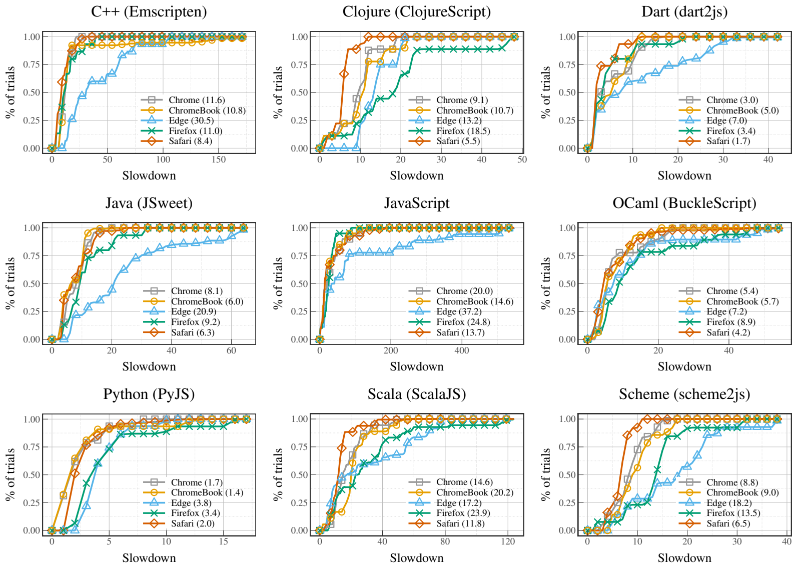
Example: Performance of PyJS





Performance relative to unmodified PyJS on a suite of 10 Python benchmarks run 10 times each. Each graph shows how an option setting affects running time or latency. Error bars show the 95% confidence interval.
Speedups are a fluke, due to PyJS generating really bad code.

Performance Evaluation



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

PyJS + Stopify vs. Skulpt

- Skulpt: implementation of Python in JavaScript that does execution control
- PyJS: implementation of Python in JavaScript without execution control
- Issues: (1) Skuplt can only run 8 of 16 benchmarks, (2) PyJS and Skulpt pass/fail different portions of the CPython test suite, (3) probably other threats to validity

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

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

Stopify + Pyret

- Joe Politz, one of Pyret's lead developers, co-authored this work
- Pyret: mostly-functional programming language that compiles to JS, self-hosting compiler, does proper tail calls, blocking I/O, REPL, animations, graceful termination, etc.
- Five years of engineering, thousands of users, still has some issues
- Our approach: modify the last phase of the Pyret compiler and portions of the runtime system to invoke Stopify, instead of doing capture/restore itself (lots of code deleted)[4]

From Pyret's runtime system (several bug fixes over the years):

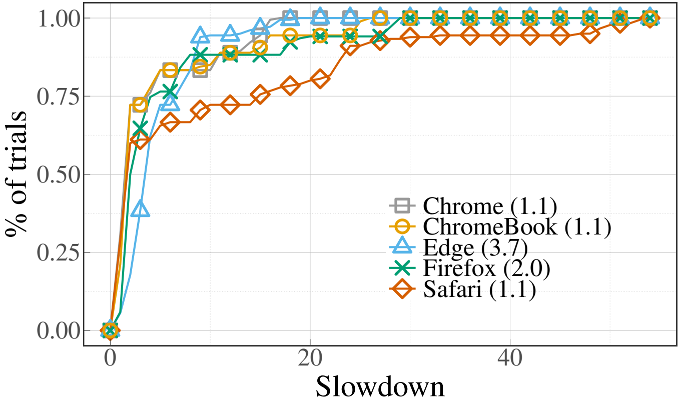
```
function eachLoop(fun, start, stop) {
  var i = start;
  function restart(_) {
    var res = thisRuntime.nothing;
    if (--thisRuntime.GAS <= 0) { res = thisRuntime.makeCont(); }
    while(!thisRuntime.isContinuation(res)) {
      if (--thisRuntime.RUNGAS <= 0) { res = thisRuntime.makeCont(); }
      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();
}
```

[4] This is the right way to use Stopify.

Stopify + Pyret

After Stopify (which generates equivalent instrumentation):

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



Conclusions

- Use existing infrastructure when possible (in this case, Babel)
- When working on a full-scale language, identify a small fragment in which the essence of the problem and solution can be formulated
- The translation may be problem-dependent (e.g., type-checking vs translation)
- ANF is likely to help you
- Take testing and CI seriously