Handlers.Js: Compiling Effect Handlers to JavaScript

Daniel Hillerström

The University of Edinburgh, UK

November 16, 2021

(joint work with Sam Lindley, KC Sivaramakrishnan, Robert Atkey, and Jeremy Yallop)

Compiling Effect Handlers to JavaScript

Compiling Effect Handlers to ~~JavaScript~~ Control Hostile Environments

Disclaimer

This work is preliminary!
(originally presented at ProWeb'18)

The "take away" slide

Implementation	**Extensions**	**Stack**	**Signature preserving**	**Translation**
CPS	None	Explicit	No	Global
Abstract machine	None	Explicit	No	Global
Generators/iterators	Generators/iterators	Implicit*	No	Global
Stack inspection	Exception handlers	Explicit (lazy)	Yes†	Global

* Trampolining requires an explicit stack representation

† Modulo effect typing

A coroutine library in one slide

A signature with a single operation $\mathrm{Yield} \overset{\mathbf{def}}{=} \{\mathrm{Yield} : \mathrm{Unit} \to \mathrm{Unit}\}$

$$\mathbf{type}\ \mathrm{Co} \overset{\mathbf{def}}{=} \mathrm{Co}(\mathrm{List}\ \mathrm{Co} \to \mathrm{Unit}!\mathrm{Yield}\})$$

$$\begin{array}{l}
\mathrm{coop} : (\mathrm{Unit} \to \mathrm{Unit}!\mathrm{Yield}) \to \mathrm{List}\ \mathrm{Co} \to \mathrm{Unit} \\
\mathrm{coop}\ p \overset{\mathbf{def}}{=} \mathbf{handle}\ p\ \langle\rangle\ \mathbf{with} \\
\qquad\qquad \{\ \mathbf{return}\ x \ \ \mapsto \lambda ps.\mathbf{case}\ ps\ \{[\,] \mapsto \langle\rangle; \mathrm{Co}\ r :: ps' \mapsto r\ \ ps'\} \\
\qquad\qquad\quad \mathrm{Yield}\ \langle\rangle\ \ k \mapsto \lambda ps.\mathbf{let}\ k' = \mathrm{Co}\,(\lambda ps.\mathbf{resume}\ r\ \mathbf{with}\ ps)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ (\mathrm{Co}\ r :: ps') = ps +\!\!+ [k']\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad r\ \ ps'\}
\end{array}$$

$$\begin{array}{l}
\mathrm{coop\_with} : (\mathrm{Unit} \to \mathrm{Unit}!\mathrm{Yield}) \to \mathrm{Co} \\
\mathrm{coop\_with}\ p \overset{\mathbf{def}}{=} \mathrm{Co}(\lambda ps.\mathrm{coop}\ \ p\ \ ps)
\end{array}$$

$$\begin{array}{l}
\mathrm{cooperate} : \mathrm{List}\,(\mathrm{Unit} \to \mathrm{Unit}!\mathrm{Yield}) \to \mathrm{Unit} \\
\mathrm{cooperate}\ rs \overset{\mathbf{def}}{=} \mathrm{coop}\ \ (\lambda\langle\rangle.\langle\rangle)\,(\mathrm{List.map}\ \mathrm{coop\_with}\ \ rs)
\end{array}$$

Kinds of effect handlers

Affine, e.g. exception handlers

$$\begin{array}{l}\{\textbf{return}\ x \mapsto \mathsf{Some}\ x \\ \ \mathsf{Fail}\ \langle\rangle\ r \mapsto \mathsf{None}\}\end{array}$$

Linear, e.g. concurrency

$$\begin{array}{l}\lambda q.\{\textbf{return}\ x \mapsto \textbf{if}\ \mathsf{Queue.is\_empty}\ q\ \textbf{then}\ \langle\rangle\ \textbf{else}\ (\mathsf{Queue.pop}\ q)\ \langle\rangle \\ \qquad \mathsf{Yield}\ \langle\rangle\ r \mapsto \mathsf{Queue.push}\ r; (\mathsf{Queue.pop}\ q)\ \langle\rangle\}\end{array}$$

Multishot, e.g. nondeterminism

$$\begin{array}{l}\{\textbf{return}\ x \mapsto [x] \\ \ \mathsf{Fork}\ \langle\rangle\ r \mapsto r\,\mathsf{true} \mathbin{+\!\!+} r\,\mathsf{false}\}\end{array}$$

Tail-resumptive, e.g. dynamic binding

$$\begin{array}{l}\lambda y.\{\textbf{return}\ x \mapsto x \\ \qquad \mathsf{Ask}\ \langle\rangle\ r \mapsto r\ y\}\end{array}$$

Three ways of handling effects

Deep capture and resumption

$$\begin{aligned}
&\textbf{handle}\ \mathcal{E}[\mathrm{op}\ V]\ \textbf{with}\ H \leadsto N[\textbf{cont}_{\langle H;\mathcal{E}\rangle}/r, V/x], && \text{where } \{\mathrm{op}\ p\ r \mapsto N\} \in H\\
&\textbf{resume}\ V\ \textbf{with}\ W \leadsto \textbf{handle}\ \mathcal{E}[W]\ \textbf{with}\ H, && \text{where } V = \textbf{cont}_{\langle H;\mathcal{E}\rangle}
\end{aligned}$$

Shallow capture and resumption

$$\begin{aligned}
&\textbf{handle}\ \mathcal{E}[\mathrm{op}\ V]\ \textbf{with}\ H \leadsto N[\textbf{cont}_{\langle \mathcal{E}\rangle}/r, V/x], && \text{where } \{\mathrm{op}\ p\ r \mapsto N\} \in H\\
&\textbf{resume}\ V\ \textbf{with}\ W \leadsto \mathcal{E}[W], && \text{where } V = \textbf{cont}_{\langle \mathcal{E}\rangle}
\end{aligned}$$

‘Sheep’ allocation, capture, and resumption

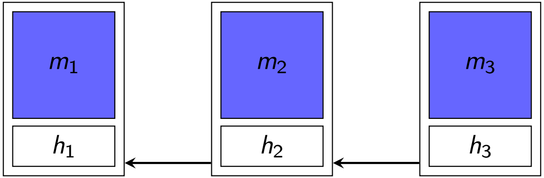
$$\begin{aligned}
&\textbf{new}\ V \leadsto \textbf{cont}_{\langle V\,[\,]\rangle}, && \text{where } V = \lambda\langle\rangle.M\\
&\underline{\textbf{handle}}\ \mathcal{E}[\mathrm{op}\ V]\ \underline{\textbf{with}}\ H \leadsto N[\textbf{cont}_{\langle \mathcal{E}\rangle}/r, V/x], && \text{where } \{\mathrm{op}\ p\ r \mapsto N\} \in H\\
&\textbf{resume}\ V\ \textbf{with}\ \langle H; W\rangle \leadsto \underline{\textbf{handle}}\ \mathcal{E}[W]\ \underline{\textbf{with}}\ H, && \text{where } V = \textbf{cont}_{\langle \mathcal{E}\rangle}
\end{aligned}$$

JavaScript: the challenges

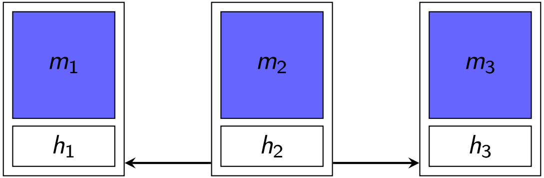
Some challenges posed by the JavaScript web runtime

- Event-driven computation: necessitates yields every-so-often
- No tail call elimination
- No access to the control state
- Difficult to predict performance

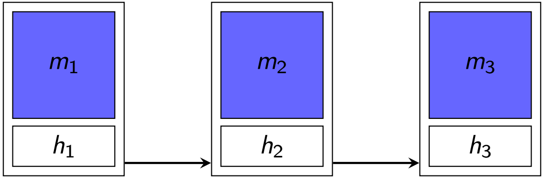
OCaml: effect handlers via segmented stacks (Sivaramakrishnan et al. 2021)



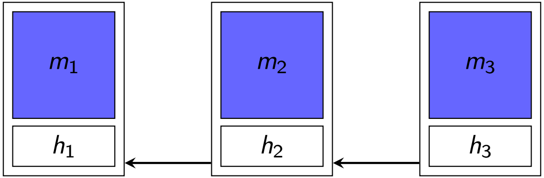
OCaml: effect handlers via segmented stacks (Sivaramakrishnan et al. 2021)



OCaml: effect handlers via segmented stacks (Sivaramakrishnan et al. 2021)



OCaml: effect handlers via segmented stacks (Sivaramakrishnan et al. 2021)



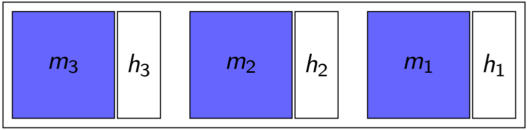
Effect handlers via generalised continuations (Hillerström et al. 2020)

Generalised continuations are a *stackless* encoding of segmented stacks.

Key idea: Represent segmented stacks as list of lists.

$$\textbf{type}\ \mathsf{GCont} \overset{\textbf{def}}{=} \mathsf{List}\ \langle \mathsf{List}\,\mathsf{Frame} \times \mathsf{Hdef} \rangle$$

Pictorially



Continuation passing style

Pros

- Well-established compilation technique
- Well-understood use of λ-calculus
- Well-optimised programming idiom in JavaScript engines

```
⟦let x = M in N⟧ = function(k) {
                     return ⟦M⟧(function(x) { return ⟦N⟧(k); });
                   }
```

Cons

- Difficult to debug (esp. with a trampoline)
- Everything gets heap allocated
- CPS transform for effect handlers is complicated (Hillerström et al. 2017)

Continuation passing style (cont'd)

Naïve idea: Compile effect handlers by argument passing, e.g.

$$\begin{aligned}
[\![\textbf{perform}\ \ell\ V]\!] &\overset{\textsf{def}}{=} \lambda k.\lambda h.h\ \langle \ell, \langle [\![V]\!], \lambda x.k\ x\ h\rangle\rangle \\
[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] &\overset{\textsf{def}}{=} \lambda k.[\![M]\!]\,(\lambda x.[\![N]\!]\ k) \\
[\![\textbf{return}\ V]\!] &\overset{\textsf{def}}{=} \lambda k.k\,[\![V]\!]
\end{aligned}$$

Continuation passing style (cont'd)

Naïve idea: Compile effect handlers by argument passing, e.g.

$$
\begin{aligned}
[\![\mathbf{perform}\ \ell\ V]\!] &\overset{\mathsf{def}}{=} \lambda k.\lambda h.h\ \langle \ell, \langle [\![V]\!], \lambda x.k\ x\ h\rangle\rangle \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] &\overset{\mathsf{def}}{=} \lambda k.[\![M]\!]\ (\lambda x.[\![N]\!]\ k) \\
[\![\mathbf{return}\ V]\!] &\overset{\mathsf{def}}{=} \lambda k.k\ [\![V]\!]
\end{aligned}
$$

The image is not *properly tail-recursive*

$$
\begin{aligned}
\top[\![\mathbf{return}\ \langle\rangle]\!] &= (\lambda k.k\ \langle\rangle)\ (\lambda x.\lambda h.x)\ (\lambda\langle z, \_\rangle.\mathbf{absurd}\ z) \\
&\rightsquigarrow ((\lambda x.\lambda h.x)\ \langle\rangle)\ (\lambda\langle z, \_\rangle.\mathbf{absurd}\ z) \\
&\rightsquigarrow (\lambda h.\langle\rangle)\ (\lambda\langle z, \_\rangle.\mathbf{absurd}\ z) \\
&\rightsquigarrow \langle\rangle
\end{aligned}
$$

(the image also contains *static administrative redexes*)

Continuation passing style (cont'd)

Naïve idea: Compile effect handlers by argument passing, e.g.

$$\begin{aligned}
[\![\textbf{perform}\ \ell\ V]\!] &\overset{\textsf{def}}{=} \lambda k.\lambda h.h\ \langle \ell, \langle [\![V]\!], \lambda x.k\ x\ h\rangle\rangle \\
[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] &\overset{\textsf{def}}{=} \lambda k.[\![M]\!]\ (\lambda x.[\![N]\!]\ k) \\
[\![\textbf{return}\ V]\!] &\overset{\textsf{def}}{=} \lambda k.k\ [\![V]\!]
\end{aligned}$$

The image is not *properly tail-recursive*

$$\begin{aligned}
\top[\![\textbf{return}\ \langle\rangle]\!] &= (\lambda k.k\ \langle\rangle)\ (\lambda x.\lambda h.x)\ (\lambda\langle z, \_\rangle.\textbf{absurd}\ z) \\
&\rightsquigarrow ((\lambda x.\lambda h.x)\ \langle\rangle)\ (\lambda\langle z, \_\rangle.\textbf{absurd}\ z) \\
&\rightsquigarrow (\lambda h.\langle\rangle)\ (\lambda\langle z, \_\rangle.\textbf{absurd}\ z) \\
&\rightsquigarrow \langle\rangle
\end{aligned}$$

(the image also contains *static administrative redexes*)

Continuation passing style (cont'd)

Solution: Uncurry the translation, e.g.

$$\begin{aligned}
[\![\textbf{perform}\ \ell\ V]\!] &\overset{\text{def}}{=} \lambda(k :: h :: ks).h\,\langle \ell, \langle [\![V]\!], \lambda x.\lambda ks'.k\,x\,(h :: ks')\rangle\rangle\, ks \\
[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] &\overset{\text{def}}{=} \lambda(k :: ks).[\![M]\!]((\lambda x.\lambda ks'.[\![N]\!](k :: ks')) :: ks) \\
[\![\textbf{return}\ V]\!] &\overset{\text{def}}{=} \lambda(k :: ks).k\,[\![V]\!]\,ks
\end{aligned}$$

Now the image is properly tail-recursive

$$\begin{aligned}
\top[\![\textbf{return}\ \langle\rangle]\!] &= (\lambda(k :: ks).k\,\langle\rangle\,ks)\,((\lambda x.\lambda ks.x) :: (\lambda\langle z, \_\rangle.\lambda ks.\textbf{absurd}\,z) :: [\,]) \\
&\rightsquigarrow (\lambda x.\lambda ks.x)\,\langle\rangle\,((\lambda\langle z, \_\rangle.\lambda ks.\textbf{absurd}\,z) :: [\,]) \\
&\rightsquigarrow \langle\rangle
\end{aligned}$$

(the image still contains static administrative redexes)

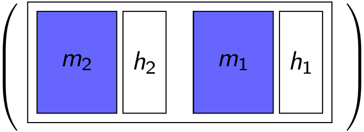
Continuation passing style (cont'd)

Long story short: Efficient one-pass CPS transform for effect handlers requires partial evaluation, e.g.

$$
\begin{aligned}
\llbracket \mathbf{perform}\ \ell\ V \rrbracket &\overset{\mathsf{def}}{=} \overline{\lambda}\overline{\langle}\theta, \overline{\langle}\xi, \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\overline{\rangle} \mathbin{\overline{::}} \kappa.\, {\downarrow}\chi^{\mathrm{ops}} \mathbin{\underline{@}} \underline{\langle}{\downarrow}\xi, \ell, \underline{\langle}\llbracket V\rrbracket, \underline{\langle}{\downarrow}\theta, \underline{\langle}{\downarrow}\xi, {\downarrow}\chi^{\mathrm{ret}}, {\downarrow}\chi^{\mathrm{ops}}\underline{\rangle}\underline{\rangle} \mathbin{\underline{::}} \underline{[\,]}\underline{\rangle}\underline{\rangle} \mathbin{\underline{@}} {\downarrow}\kappa \\
\llbracket \mathbf{let}\ x = M\ \mathbf{in}\ N \rrbracket &\overset{\mathsf{def}}{=} \overline{\lambda}\overline{\langle}\theta, \chi\overline{\rangle} \mathbin{\overline{::}} \kappa.\llbracket M \rrbracket \mathbin{\overline{@}} (\overline{\langle}{\uparrow}((\underline{\lambda} x\, ks.\mathbf{let}\ (k \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ \llbracket N\rrbracket \mathbin{\overline{@}} ({\uparrow}k \mathbin{\overline{::}} {\uparrow}ks')) \mathbin{\underline{::}} {\downarrow}\theta), \chi\overline{\rangle} \mathbin{\overline{::}} \kappa) \\
\llbracket \mathbf{return}\ V \rrbracket &\overset{\mathsf{def}}{=} \overline{\lambda}\kappa.\mathbf{app}\ ({\downarrow}\kappa)\ \llbracket V \rrbracket
\end{aligned}
$$

Intuition: Explicit passing of the control state (represented as a generalised continuation)

$\llbracket f\,\langle\rangle \rrbracket = f$



The CEK machine

$$\langle C \mid E \mid K \rangle$$

The CEK machine consists of three components

- Control, the expression being evaluated
- Environment, binding free variables
- Kontinuation, the continuation of C

The CEK machine

$$\langle C \mid E \mid K \rangle$$

The CEK machine consists of three components

- Control, the expression being evaluated
- Environment, binding free variables
- Kontinuation, the continuation of C

Classic continuation structure (Felleisen and Friedman 1987)

$$K : \text{List Frame}$$

The CEK machine

$$\langle C \mid E \mid K \rangle$$

The CEK machine consists of three components

- Control, the expression being evaluated
- Environment, binding free variables
- Kontinuation, the continuation of C

Obtain a runtime for handlers by plugging in generalised continuations (Hillerström 2021)

$$K : \mathsf{List}\,\langle \mathsf{List}\ \mathsf{Frame} \times \mathsf{Hdef} \rangle$$

CEK: abstract machine (cont'd)

Pros

- Fairly easy to debug
- Hot machine loop
- Well-understood technique

Cons

- Allocation intensive
- Everything is boxed
- Basic operations are not JIT optimised

```
let control = initial_program;
let environment = initial_env;
let kontinuation = k_identity; // generalised continuation
let config = NORMAL;

while (true) {
  switch (config) {
    case NORMAL:
      switch (match(control)) {
        // Match AST
        // ...
        continue;
      }
    case ADM_APPLY_CONT:
      // Apply ‘kontinuation‘ to the value in ‘control‘
      // ...
      continue;
  }
}
```

Generators and iterators: native JavaScript continuations

Generators and iterators provide a form of delimited control (James and Sabry 2011).
The main idea

- Transform every function into a generator
- Transform each handler into a generator that iterates its given computation

$$\begin{aligned}
\llbracket \texttt{\textbf{perform}}\ \ell\ V \rrbracket &= \texttt{\textbf{yield} \{'tag':}\ell\texttt{, 'payload':}\llbracket V \rrbracket\texttt{\}} \\
\llbracket \texttt{\textbf{let}}\ x = M\ \texttt{\textbf{in}}\ N \rrbracket &= \texttt{\textbf{var} x = }\llbracket M \rrbracket\texttt{; }\llbracket N \rrbracket \\
\llbracket \texttt{\textbf{return}}\ V \rrbracket &= \llbracket V \rrbracket
\end{aligned}$$

Pros

- Native JavaScript
- Fairly easy to debug (unless trampolined)

Cons

- Everything lives in a generator (**`function`**$*$)
- Generators/iterators use simulated continuations under-the-hood

Generators and iterators: native JavaScript continuations (cont'd)

Potential for efficient compilation of tail-resumptive handlers

$$[\![H_{\text{reader}} := \lambda v.\{\textbf{return}\ x \mapsto x; \text{Ask}\ \langle\rangle\ r \mapsto r\ v\}]\!]$$

```
= function* Hreader(v, m) {
    var g = m();
    var res = g.next(); // run until first yield
    while (!res.done) {
      switch (res.value.tag) {
      case "Ask":
        res = g.next(v); // invoke the continuation with ‘v‘
        break;
      default: // effect forwarding
        var y = yield res.value;
        res = g.next(y);
      }
    }
  }
```

Generalised stack inspection

Generalised stack inspection provides simulate continuations using exceptions (Pettyjohn et al. 2005).

- Enclose every binding in an exception handler
- Throw an exception to assemble the continuation

Pros

- Compatibility with (first-order) legacy code
- Continuation capture is on-demand

Cons

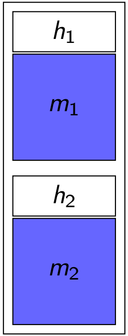
- Difficult to debug
- Deoptimisations often kick in

```
   ⟦perform ℓ V⟧ = throw new PerformOperation(ℓ, ⟦V⟧);
⟦let x = M in N⟧ = function let_x(...) {
                     var x;
                     try {
                       x = ⟦M⟧(...);
                     } catch (e) {
                       if (e instanceof PerformOperation) {
                         e.augment(⟦N⟧);
                         throw e;
                       } else {
                         throw e;
                       }
                     }
                    return ⟦N⟧(x, ...);
                  }
     ⟦return V⟧ = function(...) { return ⟦V⟧; }
```

Generalised stack inspection (cont'd)

Initially there is only the call stack

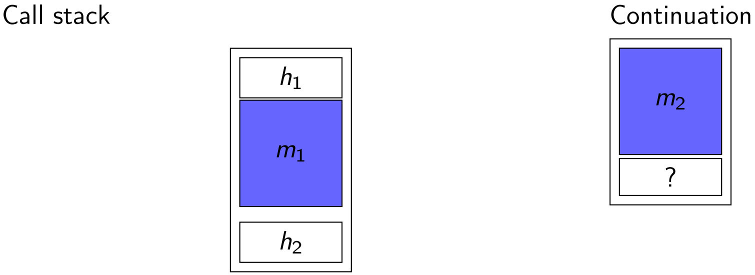
Call stack



Continuation

Generalised stack inspection (cont'd)

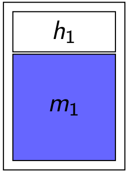
Throwing an exception causes the continuation to materialise



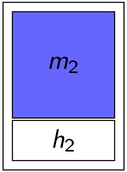
Generalised stack inspection (cont'd)

Instantiate the abstract handler once we pass over a concrete handler

Call stack



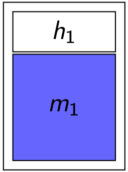
Continuation



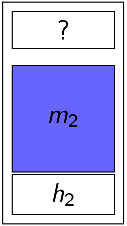
Generalised stack inspection (cont'd)

Continue unwinding the call stack

Call stack

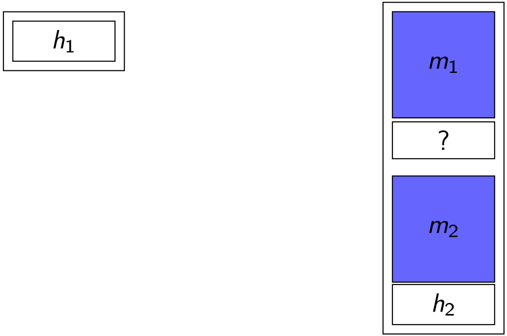


Continuation



Generalised stack inspection (cont'd)

Continue unwinding the call stack


Call stack

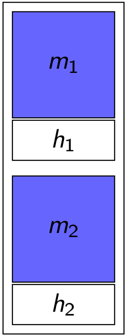
Continuation

Generalised stack inspection (cont'd)

Notice that the continuation was built in reverse

Call stack

Continuation

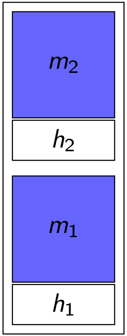


Generalised stack inspection (cont'd)

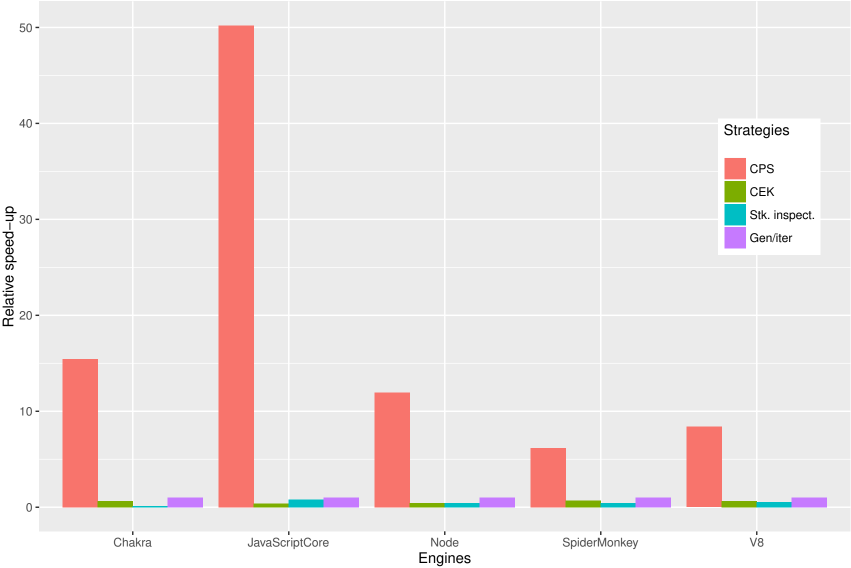
The continuation is reversed prior to invocation

Call stack

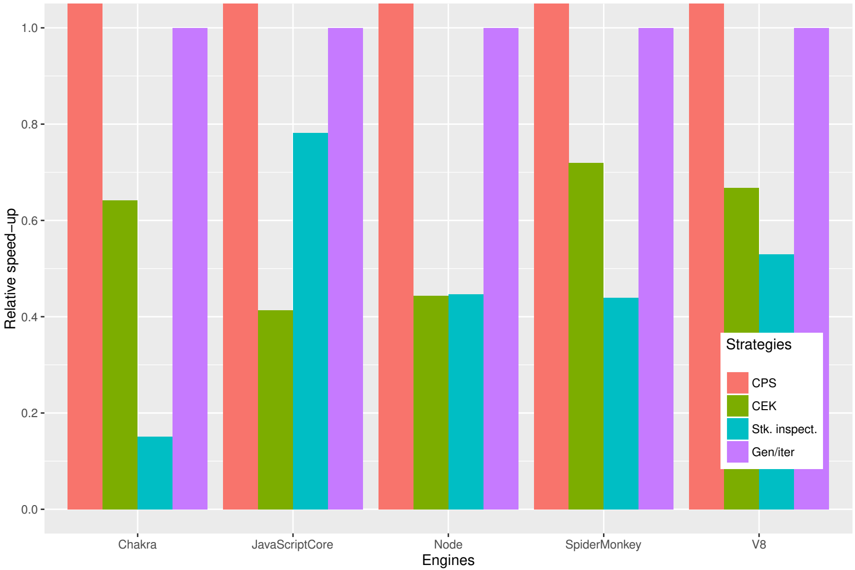
Continuation



Preliminary experiments



Preliminary experiments



Effect handlers benchmark suite

Join the effort to build a standardised benchmark suite

`https://github.com/effect-handlers/effect-handlers-bench`

Alternatives

- Selective linear CPS
  - Direct code whenever possible
  - Optimised for one-shot continuations
  - Reuse allocated frame slots
- Scheme Gambit virtual machine (Thivierge and Feeley 2012)
  - Use JavaScript as an assembly language
  - Primitive values
  - Maintains a “shadow” stack using a JavaScript array

Summary

In summary

Implementation	Extensions	Stack	Signature preserving	Translation
CPS	None	Explicit	No	Global
Abstract machine	None	Explicit	No	Global
Generators/iterators	Generators/iterators	Implicit*	No	Global
Stack inspection	Exception handlers	Explicit (lazy)	Yes[†]	Global

* Trampolining requires an explicit stack representation

[†] Modulo effect typing

References I

Felleisen, Matthias and Daniel P. Friedman (1987). “Control Operators, the SECD-machine, and the λ-Calculus”. In: *Formal Description of Programming Concepts III*, pp. 193–217.

Ganz, Steven E., Daniel P. Friedman, and Mitchell Wand (1999). “Trampolined Style”. In: *ICFP*. ACM, pp. 18–27.

Pettyjohn, Greg et al. (2005). “Continuations from generalized stack inspection”. In: *ICFP*. ACM, pp. 216–227.

James, Roshan P and Amr Sabry (2011). “Yield: Mainstream delimited continuations”. In: *TPDC*.

Thivierge, Eric and Marc Feeley (2012). “Efficient compilation of tail calls and continuations to JavaScript”. In: *Scheme@ICFP*. ACM, pp. 47–57.

Hillerström, Daniel et al. (2017). “Continuation Passing Style for Effect Handlers”. In: *FSCD*. Vol. 84. LIPIcs, 18:1–18:19.

Hillerström, Daniel, Sam Lindley, and Robert Atkey (2020). “Effect handlers via generalised continuations”. In: *J. Funct. Program.* 30, e5.

Hillerström, Daniel (2021). “Foundations for Programming and Implementing Effect Handlers”. PhD thesis. School of Informatics, The University of Edinburgh, UK.

References II

Sivaramakrishnan, KC et al. (2021). "Retrofitting Effect Handlers onto OCaml". In: *CoRR* abs/2104.00250.