Handlers.Js

A Comparative Study of Implementation Strategies for Effect Handlers on the Web

Daniel Hillerström

Laboratory for Foundations of Computer Science
School of Informatics
The University of Edinburgh, UK

April 5, 2018

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

Asynchronous trends in web programming

Call mania	Nest of callbacks
Monadic then	Chaining of promises via `then`
Star fascination	Pervasiveness of **`function`**$*$ and **`yield`**$*$
Async idiom	Ubiquity of **`async function`** and **`await`**

Asynchronous trends in web programming

Call mania Nest of callbacks

Monadic then Chaining of promises via `then`

Star fascination Pervasiveness of **`function*`** and **`yield*`**

Async idiom Ubiquity of **`async function`** and **`await`**

Effect handlers subsume all of these “idioms”

Applications of Plotkin and Pretnar (2013)'s effect handlers

Effect handlers subume contemporary control abstraction

- Generators and iterators (Leijen 2017b)
- Async/await and promises (Leijen 2017a)
- Co-routines (Kiselyov et al. 2013)

More generally, effect handlers have applied in

- Concurrency (Dolan et al. 2017)
- Multi-staging (Yallop 2017)
- Probabilistic programming (Goodman 2017)
- Backtracking (Wu et al. 2014)
- Modular program construction (Kammar et al. 2013)

. . . and this is all in *direct style!*

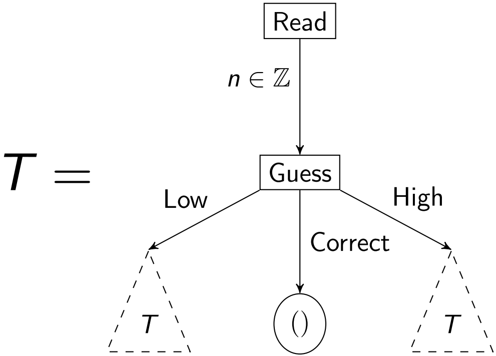
Effect handlers by example

Consider the classic “guess a number game”

```
fun game() {
  print("Take a guess>");
  var number = do Read;
  switch (do Guess(number)) {
    case Low ->
      print("Wrong: Your guess is too low.\n");
      game()
    case Correct ->
      print("Correct!!\n")
    case High ->
      print("Wrong: Your guess is too high.\n");
      game()
  }
}
```

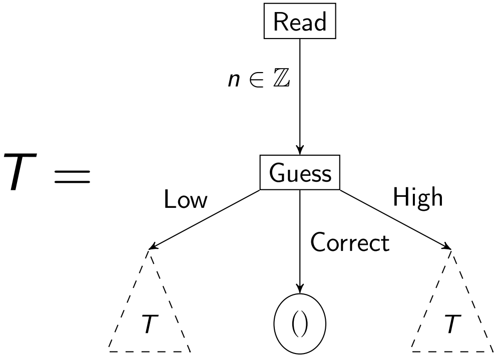
Effect handlers by example

The abstract computation induces a computation tree.



Effect handlers by example

The abstract computation induces a computation tree. We need an interpreter!



Effect handlers by example

Interpretation of `Guess` is a simple validation check

```
fun mySecret(secret, m)() {
  handle(m()) {
    case Return(x) -> x
    case Guess(n, resume) ->
      if (n < secret) resume(Low)
      else if (n > secret) resume(High)
      else resume(Correct)
  }
}
```

Effect handlers by example

Interpretation of `Guess` is a simple validation check

```
fun mySecret(secret, m)() {
  handle(m()) {
    case Return(x) -> x
    case Guess(n, resume) ->
      if (n < secret) resume(Low)
      else if (n > secret) resume(High)
      else resume(Correct)
  }
}
```

We can mock `Read` using a *parameterised* handler

```
fun input(myGuesses, m)() {
  handle(m())(myGuesses -> nextGuess) {
    case Return(_) -> ()
    case Read(resume) ->
       switch(nextGuess) {
         case [] -> ()
         case g :: gs ->
           println(intToString(g)); resume(g, gs)
       }
  }
}
```

Effect handlers by example

Plugging everything together

```
> mySecret(2, input([4,0,2], game))()
Take a guess> 4
Wrong: Your guess is too high.

Take a guess> 0
Wrong: Your guess is too low.

Take a guess> 2
Correct!!
() : ()
```

Effect handlers by example

We can modularly reinterpret operations of an abstract computation

```
fun history(m)() {
  handle(m())([] -> hist) {
    case Return(_) -> hist
    case Guess(n, resume) ->
       var an = do Guess(n);
       resume(an, (n, an) :: hist)
  }
}
```

Effect handlers by example

We can modularly reinterpret operations of an abstract computation

```
fun history(m)() {
  handle(m())([] -> hist) {
    case Return(_) -> hist
    case Guess(n, resume) ->
       var an = do Guess(n);
       resume(an, (n, an) :: hist)
  }
}
```

Plugging `history` into the pipeline yields

```
> mySecret(2, history(input([4,0,2], game)))()
(same as before)
[(2, Correct), (0, Low), (4, High)] : [(Int, Answer)]
```

Five implementation strategies

The following are feasible compilation strategies

- Free monad

  Kiselyov et al. (2013), Kammar et al. (2013), and Pretnar et al. (2017)
- Abstract machine

  Hillerström and Lindley (2016)
- Continuation-passing style

  Leijen (2017b) and Hillerström et al. (2017)
- Generators and iterators (James and Sabry 2011)
- Generalised stack inspection (Pettyjohn et al. 2005; Loitsch 2007)

Five implementation strategies

The following are feasible compilation strategies

- Free monad

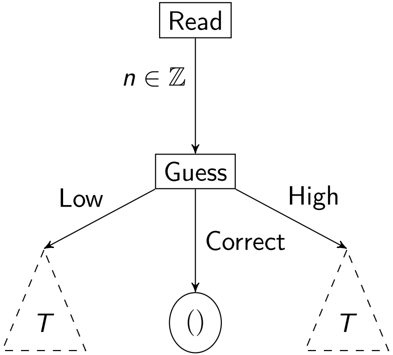
  Kiselyov et al. (2013), Kammar et al. (2013), and Pretnar et al. (2017)
- Abstract machine

  Hillerström and Lindley (2016)
- Continuation-passing style

  Leijen (2017b) and Hillerström et al. (2017)
- Generators and iterators (James and Sabry 2011)
- Generalised stack inspection (Pettyjohn et al. 2005; Loitsch 2007)

Free monad (Kiselyov et al. 2013; Kammar et al. 2013)

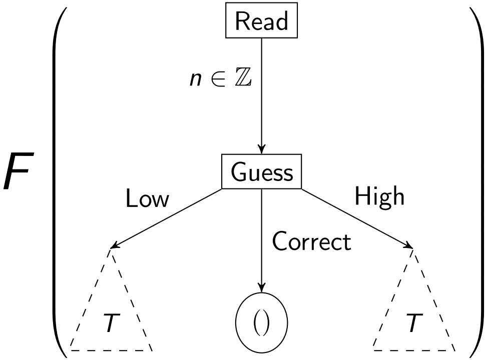
Idea: folds over computation trees.



In some sense the “standard implementation technique”.

Free monad (Kiselyov et al. 2013; Kammar et al. 2013)

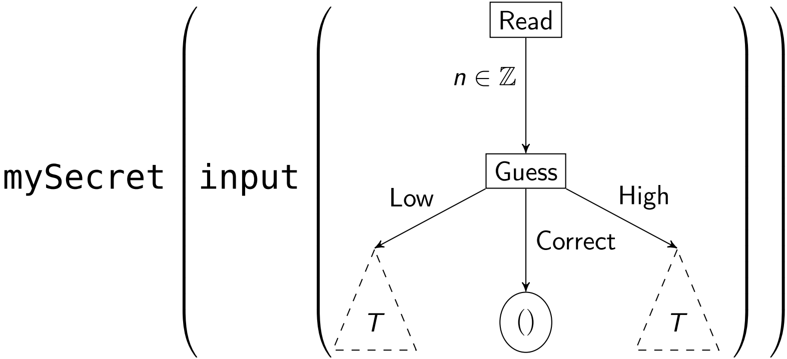
Idea: folds over computation trees.



In some sense the “standard implementation technique”.

Free monad (Kiselyov et al. 2013; Kammar et al. 2013)

Idea: folds over computation trees.



In some sense the "standard implementation technique".

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 1986)

$$K : \mathit{List}(\mathit{Frame})$$

The CEK machine

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

The CEK machine consists of three components

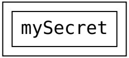
- Control, the expression being evaluated
- Environment, binding free variables
- Kontinuation, the continuation of C

With handlers, the structure gets "bumped" (Hillerström and Lindley 2016)

$$K : \mathit{List}(\mathit{Handler} \times \mathit{List}(\mathit{Frame}))$$

The CEK machine (cont'd)

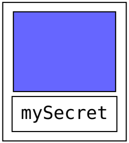
The continuation structure pictorially



mySecret

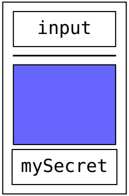
The CEK machine (cont'd)

The continuation structure pictorially



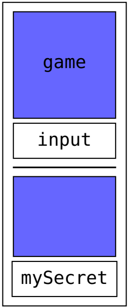
The CEK machine (cont'd)

The continuation structure pictorially



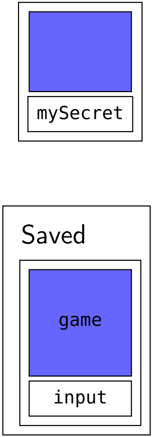
The CEK machine (cont'd)

The continuation structure pictorially



The CEK machine (cont'd)

Performing `Guess` unwinds the stack



`mySecret`

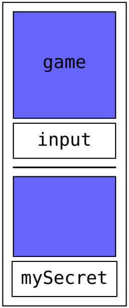
Saved

`game`

`input`

The CEK machine (cont'd)

Resuming inside `mySecret` restores the stack



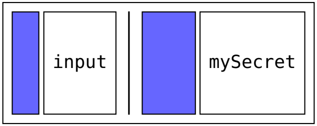
Continuation-passing style

CPS is in some sense *the* classic approach (Appel 1992; Kennedy 2007).

- Explicit control flow
- Every function call is a tail call

CPS for effect handlers (Hillerström et al. 2017)

- Use a continuation structure akin to that of the abstract machine, i.e. a stack
- Pass the stack around explicitly

$[\![\mathrm{game}()]\!] = \mathrm{game}\Bigg($  $\Bigg)$

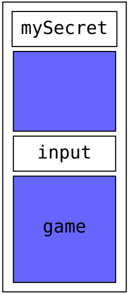
Generators and iterators

Generators and iterators provide a restricted 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

The continuation is implicit in the call stack



Generalised stack inspection

Generalised stack inspection provides a way to capture continuations using exception handlers (Pettyjohn et al. 2005).

The basic idea

- The call stack reflects the continuation
- Enclose every binding in an exception handler
- Throw an exception to assemble the continuation

Generalised stack inspection

Generalised stack inspection provides a way to capture continuations using exception handlers (Pettyjohn et al. 2005).

The basic idea

- The call stack reflects the continuation
- Enclose every binding in an exception handler
- Throw an exception to assemble the continuation

$[\![\ \mathbf{var}\ \mathrm{x} = M;\ N \ ]\!] =$

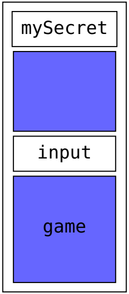
```
var x;
try {
  x = ⟦M⟧;
} catch (e) {
  if (e instanceof PerformOperation) {
    e.augment(⟦N⟧);
    throw e;
  } else {
    throw e;
  }
}
return (⟦N⟧ @ x);
```

Generalised stack inspection (cont'd)

Initially there is only the call stack

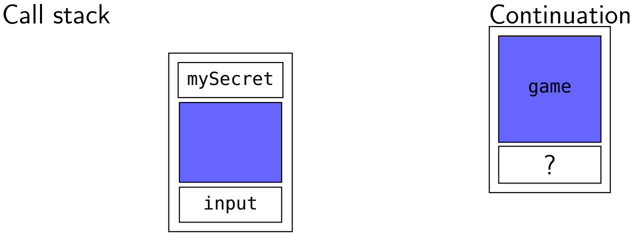
Call stack

Continuation



Generalised stack inspection (cont'd)

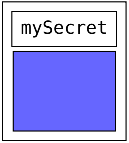
Throwing an exception causes the continuation to materialise



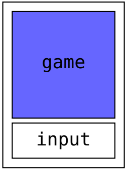
Generalised stack inspection (cont'd)

Instantiate the abstract handler once we pass over a concrete handler

Call stack

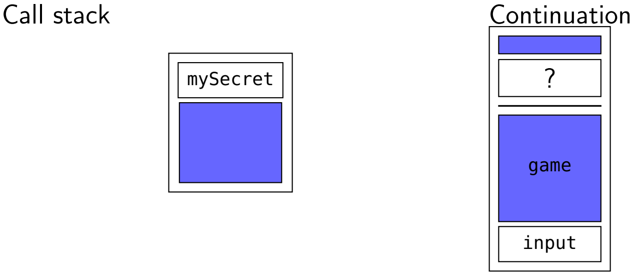


Continuation



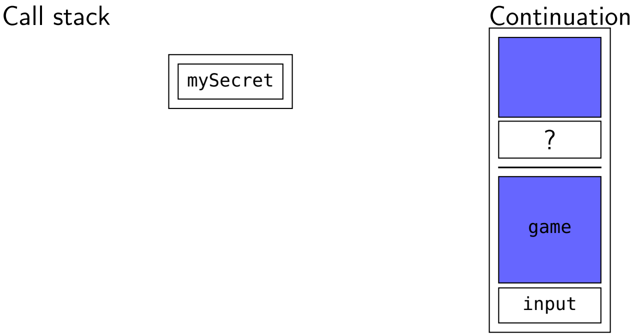
Generalised stack inspection (cont'd)

Continue unwinding the call stack



Generalised stack inspection (cont'd)

Continue unwinding the call stack

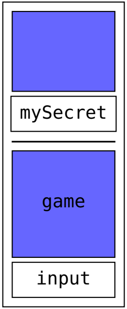


Generalised stack inspection (cont'd)

Notice that the continuation was built in reverse

Call stack

Continuation

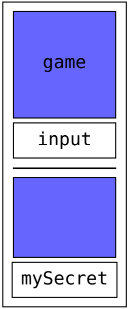


Generalised stack inspection (cont'd)

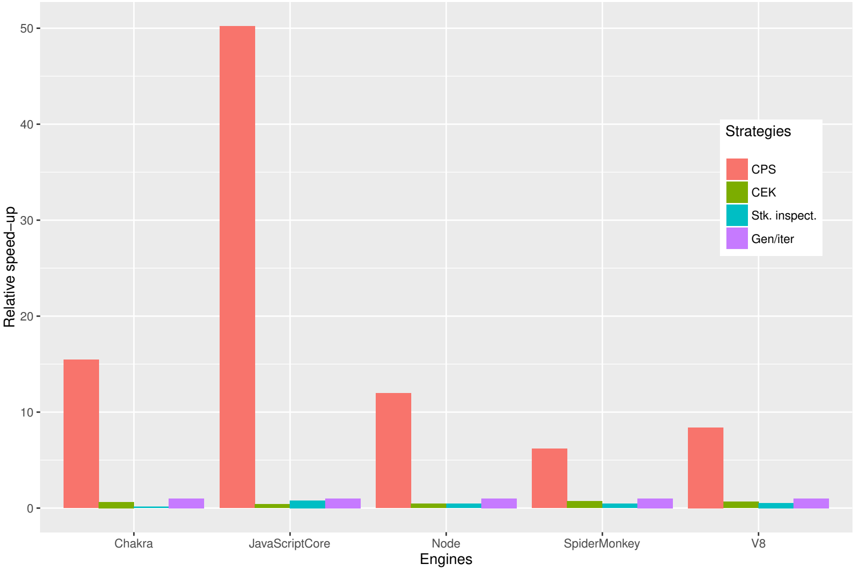
The continuation is reversed prior to invocation

Call stack

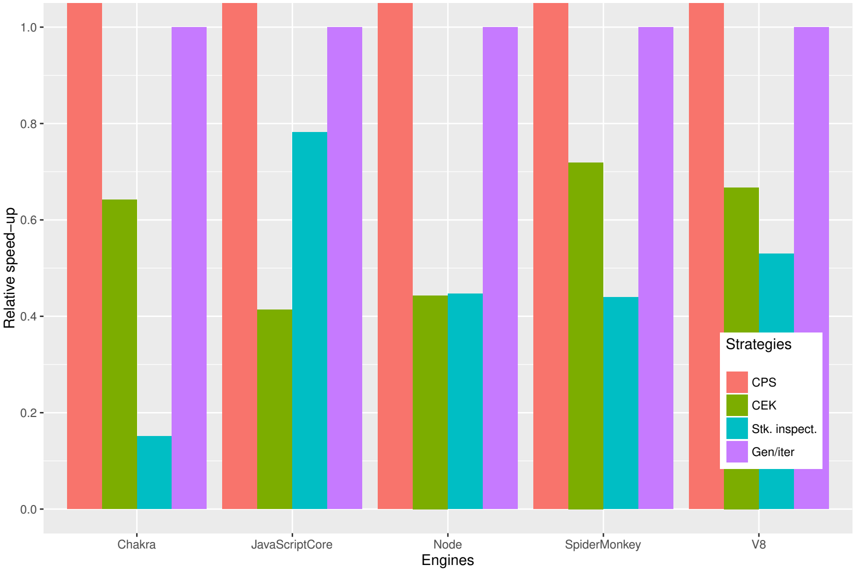
Continuation



Preliminary results



Preliminary results



Summary

Implementation	**Extensions**	**Stack**	**Type respecting**
Free monad	None	Implicit	No
Abstract machine	None	Explicit	No
CPS	None	Explicit	No
Generators/iterators	Generators/iterators	Implicit*	No
Stack inspection	Exception handlers	Explicit (lazy)	Yes[†]

* Trampolining requires an explicit stack representation

[†] Modulo effect typing

Future work

- Establish correctness of the generators/iterators and generalised stack inspection strategies
- Relate the five different compilation strategies
- More experimental evaluation

References I

Appel, Andrew W. (1992). *Compiling with Continuations*. Cambridge University Press.

Dolan, Stephen et al. (2017). "Concurrent System Programming with Effect Handlers". In: *TFP*.

Felleisen, Matthias and Daniel P. Friedman (1986). "Control Operators, the SECD-machine, and the λ-Calculus". In: *Formal Description of Programming Concepts III*. Elsevier, pp. 193–217.

Goodman, Noah (2017). *Uber AI Labs Open Sources Pyro, a Deep Probabilistic Programming Language*. URL: `https://eng.uber.com/pyro/`.

Hillerström, Daniel and Sam Lindley (2016). "Liberating effects with rows and handlers". In: *TyDe@ICFP*. ACM, pp. 15–27.

Hillerström, Daniel et al. (2017). "Continuation Passing Style for Effect Handlers". In: *FSCD*. Vol. 84. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

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

References II

Kammar, Ohad, Sam Lindley, and Nicolas Oury (2013). "Handlers in action". In: *ICFP*. ACM, pp. 145–158.

Kennedy, Andrew (2007). "Compiling with continuations, continued". In: *ICFP*. ACM, pp. 177–190.

Kiselyov, Oleg, Amr Sabry, and Cameron Swords (2013). "Extensible effects: an alternative to monad transformers". In: *Haskell*. ACM, pp. 59–70.

Leijen, Daan (2017a). "Structured Asynchrony with Algebraic Effects". In: *TyDe@ICFP*. ACM, pp. 16–29.

– (2017b). "Type directed compilation of row-typed algebraic effects". In: *POPL*. ACM, pp. 486–499.

Loitsch, Florian (2007). "Exceptional Continuations in JavaScript". In: *2007 Workshop on Scheme and Functional Programming*. Freiburg, Germany.

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

Plotkin, Gordon D. and Matija Pretnar (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4.

References III

Pretnar, Matija et al. (2017). *Efficient compilation of algebraic effects and handlers*. Tech. rep. CW 708. KU Leuven, Belgium.

Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (2014). “Effect handlers in scope”. In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, pp. 1–12. DOI: `10.1145/2633357.2633358`. URL: `http://doi.acm.org/10.1145/2633357.2633358`.

Yallop, Jeremy (2017). “Staged generic programming”. In: *PACMPL* 1.ICFP, 29:1–29:29.