Effect handlers for WebAssembly

Sam Lindley

The University of Edinburgh

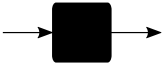
MFPS 2022

Part I

Effect handlers

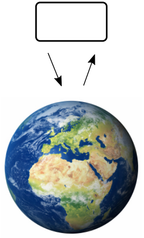
Effects

Programs as black boxes (Church-Turing model)?



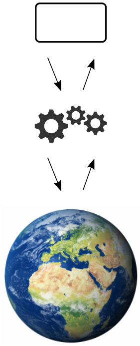
Effects

Programs must interact with their environment



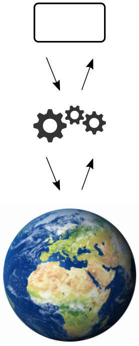
Effects

Programs must interact with their environment



Effects

Programs must interact with their environment

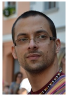


Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

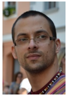
Typically ad hoc and hard-wired

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009

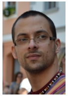
Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

Effect handlers

 Gordon Plotkin  Matija Pretnar

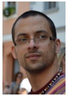
Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

Give programmer direct access to **context**

(c.f. resumable exceptions, monads, delimited control)

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

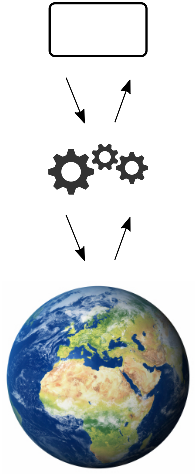
Give programmer direct access to **context**

(c.f. resumable exceptions, monads, delimited control)

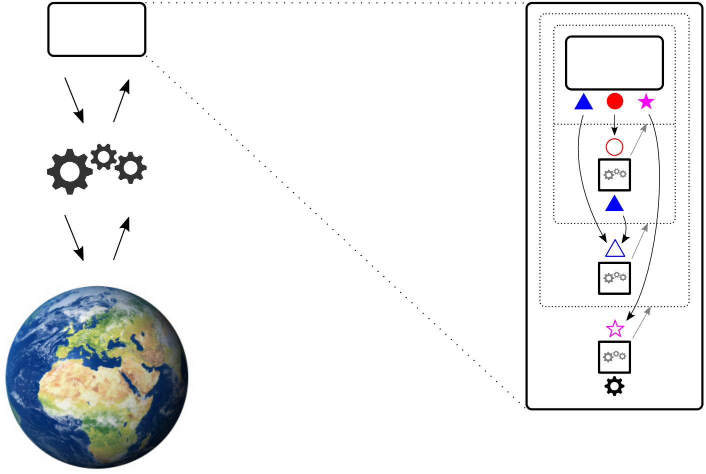
Growing industrial interest

GitHub	`semantic`	Code analysis library (> 25 million repositories)
	React	JavaScript UI library (> 2 million websites)
Uber	Pyro	Statistical inference (10% ad spend saving)

Effect handlers as composable user-defined operating systems



Effect handlers as composable user-defined operating systems



Operational semantics (deep handlers)

Reduction rules

$$
\begin{aligned}
\textbf{let } x = V \textbf{ in } N &\rightsquigarrow N[V/x] \\
\textbf{handle } V \textbf{ with } H &\rightsquigarrow N[V/x] \\
\textbf{handle } \mathcal{E}[\mathsf{op}\ V] \textbf{ with } H &\rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x.\textbf{handle } \mathcal{E}[x] \textbf{ with } H)/r], \quad \mathsf{op} \# \mathcal{E}
\end{aligned}
$$

where

$$
\begin{aligned}
H = \textbf{return}\, x \quad &\mapsto N \\
\langle \mathsf{op}_1\, p \rightarrow r \rangle &\mapsto N_{\mathsf{op}_1} \\
&\dots \\
\langle \mathsf{op}_k\, p \rightarrow r \rangle &\mapsto N_{\mathsf{op}_k}
\end{aligned}
$$

Evaluation contexts

$$
\mathcal{E} ::= [\ ] \mid \textbf{let } x = \mathcal{E} \textbf{ in } N \mid \textbf{handle } \mathcal{E} \textbf{ with } H
$$

Typing rules (deep handlers)

Effects

$$E ::= \emptyset \mid E \uplus \{\mathsf{op} : A \twoheadrightarrow B\}$$

Computations

$$C, D ::= A!E$$

Operations

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{op}\ V : B!(E \uplus \{\mathsf{op} : A \twoheadrightarrow B\})}$$

Handlers

$$\frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to D \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\mathbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r \rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to D \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\mathbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\mathbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \mathbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x.\mathbf{handle}\ \mathcal{E}[x]\ \mathbf{with}\ H)/r], \quad \mathsf{op} \# \mathcal{E}$$

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to D \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x.\textbf{handle}\ \mathcal{E}[x]\ \textbf{with}\ H)/r], \quad \mathsf{op} \# \mathcal{E}$$

The body of the resumption r reinvokes the handler

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to D \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \leadsto N_{\mathsf{op}}[V/p,\ (\lambda x.\textbf{handle}\ \mathcal{E}[x]\ \textbf{with}\ H)/r], \quad \mathsf{op} \# \mathcal{E}$$

The body of the resumption r reinvokes the handler

A deep handler performs a fold (catamorphism) on a computation tree

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to A!E \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\mathbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\mathbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \mathbf{with}\ H \ \rightsquigarrow\ N_{\mathsf{op}}[V/p, (\lambda x.\mathcal{E}[x])/r], \qquad \mathsf{op} \# \mathcal{E}$$

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to A ! E \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r \rangle \mapsto N_i)_i\end{array} : A ! E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p, (\lambda x.\mathcal{E}[x])/r], \quad \mathsf{op} \# \mathcal{E}$$

The body of the resumption r does not reinvoke the handler

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \qquad [\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad [\Gamma, p : A_i, r : B_i \to A!E \vdash N_i : D]_i}{\Gamma \vdash \begin{array}{l}\textbf{return } x \mapsto N \\ (\langle \mathsf{op}_i\, p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle } \mathcal{E}[\mathsf{op}\ V] \textbf{ with } H \ \rightsquigarrow N_{\mathsf{op}}[V/p, (\lambda x.\mathcal{E}[x])/r], \quad \mathsf{op} \# \mathcal{E}$$

The body of the resumption r does not reinvoke the handler

A shallow handler performs a case-split on a computation tree

Sheep effect handlers — a hybrid of shallow and deep handlers

$$\frac{[\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad \begin{array}{c}\Gamma, x : A \vdash N : D \\ [\Gamma, p : A_i, r : B_i \to (A!E \Rightarrow D) \to D \vdash N_i : D]_i\end{array}}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x\, h.\textbf{handle}\ \mathcal{E}[x]\ \textbf{with}\ h)/r], \quad \mathsf{op} \mathbin{\#} \mathcal{E}$$

Sheep effect handlers — a hybrid of shallow and deep handlers

$$\frac{[\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad \begin{array}{c}\Gamma, x : A \vdash N : D \\ [\Gamma, p : A_i, r : B_i \to (A!E \Rightarrow D) \to D \vdash N_i : D]_i\end{array}}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x\, h.\textbf{handle}\ \mathcal{E}[x]\ \textbf{with}\ h)/r], \quad \mathsf{op} \# \mathcal{E}$$

Like a shallow handler, the body of the resumption need not reinvoke the same handler

Sheep effect handlers — a hybrid of shallow and deep handlers

$$\frac{[\mathsf{op}_i : A_i \twoheadrightarrow B_i \in E]_i \qquad \begin{array}{c}\Gamma, x : A \vdash N : D \\ [\Gamma, p : A_i, r : B_i \to (A!E \Rightarrow D) \to D \vdash N_i : D]_i\end{array}}{\Gamma \vdash \begin{array}{l}\textbf{return}\ x \mapsto N \\ (\langle \mathsf{op}_i\ p \to r\rangle \mapsto N_i)_i\end{array} : A!E \Rightarrow D}$$

$$\textbf{handle}\ \mathcal{E}[\mathsf{op}\ V]\ \textbf{with}\ H \rightsquigarrow N_{\mathsf{op}}[V/p,\ (\lambda x\, h.\textbf{handle}\ \mathcal{E}[x]\ \textbf{with}\ h)/r], \quad \mathsf{op} \# \mathcal{E}$$

Like a shallow handler, the body of the resumption need not reinvoke the same handler

Like a deep handler, the body of the resumption must invoke *some* handler

Example: lightweight threads

Effect signature

$$\{\text{yield} : 1 \twoheadrightarrow 1\}$$

Example: lightweight threads

Effect signature

$$\{\text{yield} : 1 \twoheadrightarrow 1\}$$

Two cooperative lightweight threads

$$\begin{aligned}
\text{tA}\,() &= \text{print}\,(\text{“A1 ”}); \text{yield}\,(); \text{print}\,(\text{“A2 ”}) \\
\text{tB}\,() &= \text{print}\,(\text{“B1 ”}); \text{yield}\,(); \text{print}\,(\text{“B2 ”})
\end{aligned}$$

Example: lightweight threads (deep handlers)

Types

$$\text{Thread } E = 1 \to 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \qquad \text{Res } E = 1 \to \text{List } (\text{Res } E) \to 1!E$$

Handler

$$\text{coop} : 1!(\text{Thread } E) \Rightarrow (\text{List } (\text{Res } E) \to 1!E)$$

$$\begin{aligned}
\text{coop} = \textbf{return}\,() &\mapsto \lambda rs.\textbf{case } rs \textbf{ of } [] \mapsto () \\
&\qquad\qquad\qquad\quad (r :: rs) \mapsto r\ ()\ rs \\
\langle \text{yield}\,() \to s\rangle &\mapsto \lambda rs.\textbf{case } rs \textbf{ of } [] \mapsto s\ ()\ [] \\
&\qquad\qquad\qquad\quad (r :: rs) \mapsto r\ ()\ (rs \mathbin{+\!\!+} [s])
\end{aligned}$$

$$\begin{aligned}
&\text{lift} : \text{Thread } E \to \text{Res } E && \text{cooperate} : \text{List } (\text{Thread } E) \to 1!E \\
&\text{lift } t = \lambda().\textbf{handle } t() \textbf{ with } \text{coop} && \text{cooperate } ts = \text{lift id } ()\ (\text{map lift } ts)
\end{aligned}$$

Example: lightweight threads (deep handlers)

Types

$$\mathsf{Thread}\ E = 1 \to 1!(E \uplus \{\mathsf{yield} : 1 \twoheadrightarrow 1\}) \qquad \mathsf{Res}\ E = 1 \to \mathsf{List}\ (\mathsf{Res}\ E) \to 1!E$$

Handler

$$\mathsf{coop} : 1!(\mathsf{Thread}\ E) \Rightarrow (\mathsf{List}\ (\mathsf{Res}\ E) \to 1!E)$$

$$\begin{aligned}
\mathsf{coop} = \mathbf{return}\,() &\mapsto \lambda rs.\mathbf{case}\ rs\ \mathbf{of}\ [] \mapsto () \\
&\qquad\qquad\qquad (r :: rs) \mapsto r\ ()\ rs \\
\langle \mathsf{yield}\,() \to s \rangle &\mapsto \lambda rs.\mathbf{case}\ rs\ \mathbf{of}\ [] \mapsto s\ ()\ [] \\
&\qquad\qquad\qquad (r :: rs) \mapsto r\ ()\ (rs +\!\!+ [s])
\end{aligned}$$

$$\mathsf{lift} : \mathsf{Thread}\ E \to \mathsf{Res}\ E \qquad \mathsf{cooperate} : \mathsf{List}\ (\mathsf{Thread}\ E) \to 1!E$$

$$\mathsf{lift}\ t = \lambda().\mathbf{handle}\ t()\ \mathbf{with}\ \mathsf{coop} \qquad \mathsf{cooperate}\ ts = \mathsf{lift}\ \mathsf{id}\ ()\ (\mathsf{map}\ \mathsf{lift}\ ts)$$

$$\mathsf{cooperate}\,[tA, tB] \Longrightarrow ()$$

A1 B1 A2 B2

Example: lightweight threads (shallow handler)

Types

$$\text{Thread } E = 1 \to 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \qquad \text{Res } E = \text{Thread } E$$

Handler

$$\text{cooperate} : \text{List}\,(\text{Thread } E) \to 1!E$$

$$\text{cooperate}\,[] = () \qquad \begin{array}{l} \text{cooperate}\,(r :: rs) = \textbf{handle}\; r()\;\textbf{with} \\ \quad \textbf{return}\,() \qquad\quad \mapsto \text{cooperate}\,(rs) \\ \quad \langle \text{yield}\,() \to s\rangle \mapsto \text{cooperate}\,(rs +\!\!+ [s]) \end{array}$$

Example: lightweight threads (shallow handler)

Types

$$\text{Thread } E = 1 \to 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \qquad \text{Res } E = \text{Thread } E$$

Handler

$$\text{cooperate} : \text{List}\,(\text{Thread } E) \to 1!E$$

$$\text{cooperate}\,[] = () \qquad \begin{aligned} \text{cooperate}\,(r :: rs) = {} & \textbf{handle } r()\ \textbf{with} \\ & \textbf{return}\,() \mapsto \text{cooperate}\,(rs) \\ & \langle \text{yield}\,() \to s \rangle \mapsto \text{cooperate}\,(rs +\!\!+ [s]) \end{aligned}$$

$$\text{cooperate}\,[tA, tB] \Longrightarrow ()$$

A1 B1 A2 B2

Example: lightweight threads (sheep handler)

Types

$$\text{Thread } E = 1 \to 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \qquad \text{Res } E = 1 \to \text{List}\,(\text{Res } E) \to 1!E$$

Handler

$$\text{coop} : \text{List}\,(\text{Res } E) \to 1!(\text{Thread } E) \Rightarrow 1!E$$

$$\begin{array}{ll}
\text{coop}\,[\,] = & \text{coop}\,(r :: rs) = \\
\quad \textbf{return}\,() \qquad \mapsto () & \quad \textbf{return}\,() \qquad \mapsto r\ ()\ (\text{coop}\ rs) \\
\quad \langle \text{yield}\,() \to r \rangle \mapsto r\ ()\ (\text{coop}\ [\,]) & \quad \langle \text{yield}\,() \to s \rangle \mapsto r\ ()\ (\text{coop}\ (rs +\!\!+ [s]))
\end{array}$$

$$\begin{array}{ll}
\text{lift} : \text{Thread } E \to \text{Res } E & \text{cooperate} : \text{List}\,(\text{Thread } E) \to 1!E \\
\text{lift } t = \lambda()\ rs.\textbf{handle}\ t()\ \textbf{with}\ \text{coop}\ rs & \text{cooperate}\ ts = \text{lift id}\ ()\ (\text{map lift}\ ts)
\end{array}$$

Example: lightweight threads (sheep handler)

Types

$$\text{Thread } E = 1 \to 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \qquad \text{Res } E = 1 \to \text{List}\,(\text{Res } E) \to 1!E$$

Handler

$$\text{coop} : \text{List}\,(\text{Res } E) \to 1!(\text{Thread } E) \Rightarrow 1!E$$

$$\begin{array}{lll} \text{coop}\,[\,] = & & \\ \quad \textbf{return}\,() & \mapsto & () \\ \quad \langle \text{yield}\,() \to r \rangle & \mapsto & r\,()\,(\text{coop}\,[\,]) \end{array} \qquad \begin{array}{lll} \text{coop}\,(r :: rs) = & & \\ \quad \textbf{return}\,() & \mapsto & r\,()\,(\text{coop}\ rs) \\ \quad \langle \text{yield}\,() \to s \rangle & \mapsto & r\,()\,(\text{coop}\,(rs \mathbin{+\!\!+} [s])) \end{array}$$

$$\begin{array}{l} \text{lift} : \text{Thread } E \to \text{Res } E \\ \text{lift } t = \lambda()\ rs.\textbf{handle}\ t()\ \textbf{with}\ \text{coop}\ rs \end{array} \qquad \begin{array}{l} \text{cooperate} : \text{List}\,(\text{Thread } E) \to 1!E \\ \text{cooperate } ts = \text{lift id}\,()\,(\text{map lift } ts) \end{array}$$

$$\text{cooperate}\,[tA, tB] \Longrightarrow ()$$

A1 B1 A2 B2

Part II

WebAssembly with effect handlers

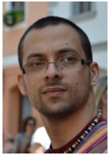
WebAssembly

Low-level language and execution environment with a formal semantics

Conceived as a target language for the web supported by all of the main web browsers

Also used e.g. for content delivery networks, library sandboxing, smart contracts

Effect handlers for WebAssembly

      . . .

(Daniel Hillerström, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, KC Sivamarakrishnan)

WasmFX (also known as "typed continuations"; implementation of "stack switching")

`https://wasmfx.dev`

Features: explicit continuation type, linear continuations, handling built into resuming, supports reference counting

Key ingredients

Continuation types

cont ⟨*typeidx*⟩	define a new continuation type

Control tags

tag ⟨*tagidx*⟩	define a new tag

Core instructions

cont.new ⟨*typeidx*⟩	create a new continuation
suspend ⟨*tagidx*⟩	suspend the current continuation
resume (**tag** ⟨*tagidx*⟩ ⟨*labelidx*⟩)∗	resume a continuation

Key ingredients

Continuation types

cont ⟨*typeidx*⟩	define a new continuation type

Control tags

tag ⟨*tagidx*⟩	define a new tag

Core instructions

cont.new ⟨*typeidx*⟩	create a new continuation
suspend ⟨*tagidx*⟩	suspend the current continuation
resume (**tag** ⟨*tagidx*⟩ ⟨*labelidx*⟩)∗	resume a continuation

Additional instructions

cont.bind ⟨*typeidx*⟩	bind a continuation to (partial) arguments
resume_throw ⟨*tagidx*⟩	abort a continuation
barrier ⟨*blocktype*⟩ ⟨*instr*⟩∗	block suspension

Control tags

Synonyms: operation, command, resumable exception, event

tag $\$e$ (**param** $s*$) (**result** $t*$)	declare tag of type $[s*] \to [t*]$
suspend $\$e : [s*] \to [t*]$	invoke tag
where e is a tag of type $[s*] \to [t*]$	

Continuations

Synonyms: stacklet, resumption

cont.new $\$ct$: $[(\textbf{ref } \$ft)] \to [(\textbf{ref } \$ct)]$ — new continuation from function

where $\$ft$ denotes a function type $[s*] \to [t*]$

$\$ct = \textbf{cont } \$ft$

resume $(\textbf{tag } \$e\ \$l)*$: $[t1*\ (\textbf{ref } \$ct)] \to [t2*]$ — invoke continuation with handler

where $\$ct = \textbf{cont } ([t1*] \to [t2*])$

each $\$e$ is a control tag and

each $\$l$ is a label pointing to its handler clause

if $\$e : [s1*] \to [s2*]$ then

$\$l : [s1*\ (\textbf{ref } \$ct')] \to [t2*]$

$\$ct' : [s2*] \to [t2*]$

Continuations

Synonyms: stacklet, resumption

cont.new $\$ct$: $[(\textbf{ref}\ \$ft)] \to [(\textbf{ref}\ \$ct)]$ — new continuation from function

where $\$ft$ denotes a function type $[s*] \to [t*]$

$\$ct = \textbf{cont}\ \$ft$

resume $(\textbf{tag}\ \$e\ \$l)*$: $[t1*\ (\textbf{ref}\ \$ct)] \to [t2*]$ — invoke continuation with handler

where $\$ct = \textbf{cont}\ ([t1*] \to [t2*])$

each $\$e$ is a control tag and

each $\$l$ is a label pointing to its handler clause

if $\$e : [s1*] \to [s2*]$ then

$\$l : [s1*\ (\textbf{ref}\ \$ct')] \to [t2*]$

$\$ct' : [s2*] \to [t2*]$

resume_throw $\$exn$: $[s*\ (\textbf{ref}\ \$ct)] \to [t2*]$ — discard cont. and throw exception

where $\$ct = \textbf{cont}\ ([t1*] \to [t2*])$

$\$exn : [s*] \to []$

Encoding handlers with blocks and labels

If $\$ei : [si*] \to [ti*]$ and $\$cti : [ti*] \to [tr*]$ then a typical handler looks something like:

```
(loop $l
  (block $on_e1 (result s1* (ref $ct1))
        ...
    (block $on_en (result sn* (ref $ctn))
      (resume
        (tag $e1 $on_e1) ... (tag $en $on_en)
        (local.get $nextk))
      ... (br $l)
    )  ;;    $on_en (result sn* (ref $ctn))
    ... (br $l)
        ...
  )  ;;    $on_e1 (result s1* (ref $ct1))
  ... (br $l))
```

- Structured as a scheduler loop
- Handler body comes *after* block
- Result specifies types of parameters and continuation

Example: lightweight threads

```
(loop $l (if (ref.is_null (local.get $nextk)) (then (return)))
  (block $on_yield (result (ref $cont))
    (block $on_fork (result (ref $cont) (ref $cont))
      (resume (tag $yield $on_yield) (tag $fork $on_fork)
              (local.get $nextk))
      (local.set $nextk (call $dequeue))
      (br $l)
    )  ;;   $on_fork (result (ref $cont) (ref $cont))
    (local.set $nextk)  ;; current thread
    (call $enqueue)  ;; new thread
    (br $l)
  )  ;;   $on_yield (result (ref $cont))
  (call $enqueue)  ;; current thread
  (local.set $nextk (call $dequeue))  ;; next thread
  (br $l))
```

Examples

Lightweight threads

Actors

Async/await

...

https://github.com/effect-handlers/wasm-spec/tree/examples/proposals/continuations/examples

Partial continuation application

No need to do any allocation as continuations are one-shot

$$\begin{array}{l}\textbf{cont.bind}\ \$ct : [s1{*}\ (\textbf{ref}\ \$ct')] \to [(\textbf{ref}\ \$ct)] \\ \quad \text{where } \$ct = \textbf{cont}\ ([s2{*}] \to [t1{*}]) \\ \qquad\qquad \$ct' = \textbf{cont}\ ([s1{*}\ s2{*}] \to [t1{*}])\end{array}$$

Partial continuation application

No need to do any allocation as continuations are one-shot

$$\begin{array}{l}\textbf{cont.bind}\ \$ct : [s1*\ (\textbf{ref}\ \$ct')] \to [(\textbf{ref}\ \$ct)] \\ \quad \text{where } \$ct = \textbf{cont}\ ([s2*] \to [t1*]) \\ \qquad\qquad \$ct' = \textbf{cont}\ ([s1*\ s2*] \to [t1*])\end{array}$$

Avoids code duplication

Barriers

Behaves like a catch-all handler that traps on suspension

$$\begin{aligned} &\textbf{barrier}\ \$l\ \$bt\ instr* : [s*] \rightarrow [t*] \\ &\quad \text{where } \$bt = [s*] \rightarrow [t*] \\ &\qquad\qquad instr* : [s*] \rightarrow [t*] \end{aligned}$$

Status

Reference interpreter extension
https://github.com/effect-handlers/wasm-spec/tree/master/interpreter

Formal spec
https://github.com/WebAssembly/stack-switching/tree/main/proposals/continuations/Overview.md

Examples
https://github.com/WebAssembly/stack-switching/tree/main/proposals/continuations/examples

What next?

Mechanise the spec

Wasmtime implementation

WasmFX backends: Links, Koka, JavaScript, Lumen, ...

Benchmarking

Potential extensions: named handlers, multishot continuations, handler return clauses, tail-resumptive handlers, first-class tags, preemption

Part III

Extensions

Named handlers

Motivation: support capability-passing style; avoid dynamic binding / dynamic scope

Named handlers

Motivation: support capability-passing style; avoid dynamic binding / dynamic scope

New reference type for handlers (unique *prompt* as in multi-prompt delimited control)

$$\textbf{handler}\ t*$$

Named handlers

Motivation: support capability-passing style; avoid dynamic binding / dynamic scope

New reference type for handlers (unique *prompt* as in multi-prompt delimited control)

$$\textbf{handler}\ t*$$

Suspending to a named handler by passing a prompt

$$\begin{array}{c}\textbf{suspend\_to}\ \$e : [s*\ (\textbf{ref}\ \$ht)] \to [t*] \\ \text{where}\ \$ht = \textbf{handler}\ tr* \\ \$e = [s*] \to [t*]\end{array}$$

Named handlers

Motivation: support capability-passing style; avoid dynamic binding / dynamic scope

New reference type for handlers (unique *prompt* as in multi-prompt delimited control)

$$\textbf{handler}\ t*$$

Suspending to a named handler by passing a prompt

$$\begin{aligned} &\textbf{suspend\_to}\ \$e : [s*\ (\textbf{ref}\ \$ht)] \to [t*] \\ &\quad \text{where } \$ht = \textbf{handler}\ tr* \\ &\qquad \$e = [s*] \to [t*] \end{aligned}$$

Resuming with a unique prompt for the handler

$$\begin{aligned} &\textbf{resume\_with}\ (\textbf{tag}\ \$e\ \$l)* : [t1*\ (\textbf{ref}\ \$ct)] \to [t2*] \\ &\quad \text{where } \$ht = \textbf{handler}\ t2* \\ &\qquad \$ct = \textbf{cont}\ ([(\textbf{ref}\ \$ht)\ t1*] \to [t2*]) \end{aligned}$$

Direct switching

Motivation: avoid a double stack-switch to implement a context switch

Direct switching

Motivation: avoid a double stack-switch to implement a context switch

Switch directly to another continuation

$$
\begin{aligned}
&\textbf{switch\_to} : [\mathit{t1}{*}\ (\textbf{ref}\ \$\mathit{ct1})\ (\textbf{ref}\ \$\mathit{ht})] \to [\mathit{t2}{*}] \\
&\quad \text{where } \$\mathit{ht} = \textbf{handler}\ \mathit{t3}{*} \\
&\qquad\qquad \$\mathit{ct1} = \textbf{cont}\ ([(\textbf{ref}\ \$\mathit{ht})\ (\textbf{ref}\ \$\mathit{ct2})\ \mathit{t1}{*}] \to [\mathit{t3}{*}]) \\
&\qquad\qquad \$\mathit{ct2} = \textbf{cont}\ ([\mathit{t2}{*}] \to [\mathit{t3}{*}])
\end{aligned}
$$

Direct switching

Motivation: avoid a double stack-switch to implement a context switch

Switch directly to another continuation

$$\begin{aligned}&\textbf{switch\_to} : [\mathit{t1}{*}\ (\textbf{ref}\ \$\mathit{ct1})\ (\textbf{ref}\ \$\mathit{ht})] \to [\mathit{t2}{*}]\\ &\quad\text{where } \$\mathit{ht} = \textbf{handler}\ \mathit{t3}{*}\\ &\qquad\qquad \$\mathit{ct1} = \textbf{cont}\ ([(\textbf{ref}\ \$\mathit{ht})\ (\textbf{ref}\ \$\mathit{ct2})\ \mathit{t1}{*}] \to [\mathit{t3}{*}])\\ &\qquad\qquad \$\mathit{ct2} = \textbf{cont}\ ([\mathit{t2}{*}] \to [\mathit{t3}{*}])\end{aligned}$$

Behaves as if we had a built-in tag

$$\textbf{tag}\ \$\mathit{switch}\ (\textbf{param}\ \mathit{t1}{*}\ (\textbf{ref}\ \$\mathit{ct1}))\ (\textbf{result}\ \mathit{t3}{*})$$

and the handler implicitly handles $\$\mathit{switch}$ by resuming to the continuation argument.

Direct switching

Motivation: avoid a double stack-switch to implement a context switch

Switch directly to another continuation

$$\begin{array}{l}\textbf{switch\_to} : [\mathit{t1}{*}\ (\textbf{ref}\ \$\mathit{ct1})\ (\textbf{ref}\ \$\mathit{ht})] \to [\mathit{t2}{*}] \\ \quad \text{where } \$\mathit{ht} = \textbf{handler}\ \mathit{t3}{*} \\ \qquad\qquad \$\mathit{ct1} = \textbf{cont}\ ([(\textbf{ref}\ \$\mathit{ht})\ (\textbf{ref}\ \$\mathit{ct2})\ \mathit{t1}{*}] \to [\mathit{t3}{*}]) \\ \qquad\qquad \$\mathit{ct2} = \textbf{cont}\ ([\mathit{t2}{*}] \to [\mathit{t3}{*}])\end{array}$$

Behaves as if we had a built-in tag

$$\textbf{tag}\ \$\mathit{switch}\ (\textbf{param}\ \mathit{t1}{*}\ (\textbf{ref}\ \$\mathit{ct1}))\ (\textbf{result}\ \mathit{t3}{*})$$

and the handler implicitly handles $*switch* by resuming to the continuation argument.

In practice requires recursive types (typically $*ct1* and $*ct2* will be the same type)

Multishot continuations

Motivation: backtracking search, ProbProg, AD, etc.

Multishot continuations

Motivation: backtracking search, ProbProg, AD, etc.

Easy to adapt the formal semantics to not trap when a continuation is used twice...
...but would seem to preclude expected implementation techniques!

Multishot continuations

Motivation: backtracking search, ProbProg, AD, etc.

Easy to adapt the formal semantics to not trap when a continuation is used twice...
...but would seem to preclude expected implementation techniques!

Clone a continuation

$$\textbf{cont.clone}\ \$ct : [(\textbf{ref}\ \$ct)] \to [(\textbf{ref}\ \$ct)]$$
$$\text{where } \$ct = \textbf{cont}\ ([s*] \to [t*])$$

Multishot continuations

Motivation: backtracking search, ProbProg, AD, etc.

Easy to adapt the formal semantics to not trap when a continuation is used twice...
...but would seem to preclude expected implementation techniques!

Clone a continuation

$$\textbf{cont.clone}\ \$ct : [(\textbf{ref}\ \$ct)] \to [(\textbf{ref}\ \$ct)]$$
$$\text{where } \$ct = \textbf{cont}\ ([s*] \to [t*])$$

Alternative design: build **cont.clone** into a special variant of **resume**

Some other extensions

- handler return clauses (functional programming)
- tail-resumptive handlers (dynamic binding)
- first-class tags (modularity)
- parametric tags (existential types)
- preemption (interrupts)