Effect handlers for WebAssembly

Sam Lindley

The University of Edinburgh

26th July 2022

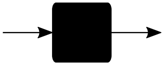
Formal Wasm meeting, Cambridge

Part I

Effect handlers

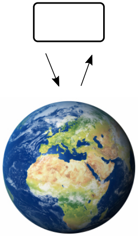
Effects

Programs as black boxes (Church-Turing model)?



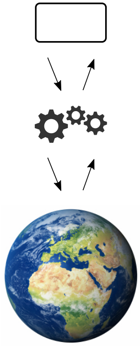
Effects

Programs must interact with their environment



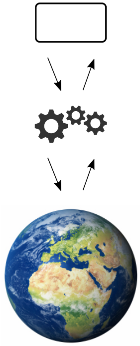
Effects

Programs must interact with their environment



Effects

Programs must interact with their environment

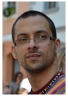


Effects are pervasive

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

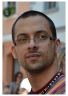
Typically ad hoc and hard-wired

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009

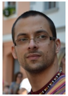
Effect handlers

 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

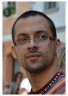
Handlers of algebraic effects, ESOP 2009

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

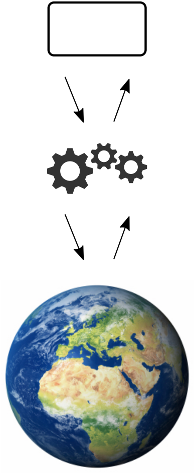
Give programmer direct access to **context**

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

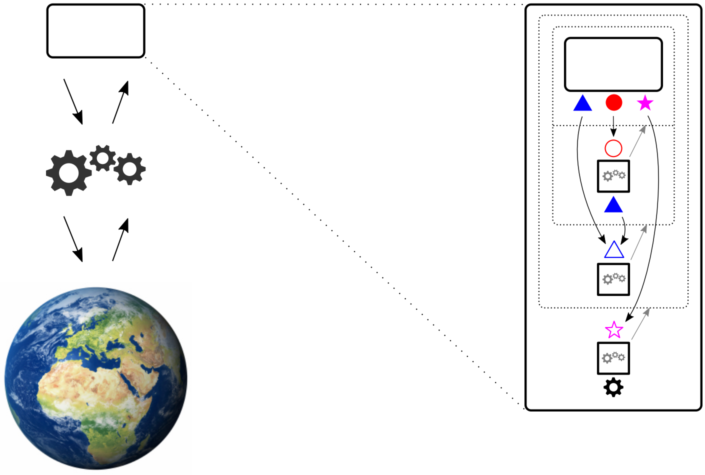
Growing industrial interest

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

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

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

Deep effect handlers

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

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

The body of the resumption r reinvokes the handler

Deep effect handlers

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

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

The body of the resumption r reinvokes the handler

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

Shallow effect handlers

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

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

Shallow effect handlers

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

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

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

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

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

Sheep effect handlers — a hybrid of shallow and deep handlers

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

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

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

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

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

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

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

Example: lightweight threads

Effect signature

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

Two cooperative lightweight threads

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

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

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

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

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

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

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

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

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

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

A1 B1 A2 B2

Example: lightweight threads (shallow handler)

Types

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

Handler

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

$$\mathrm{cooperate}\,[\,] = () \qquad \begin{array}{l} \mathrm{cooperate}\,(r :: rs) = \mathbf{handle}\ r()\ \mathbf{with} \\ \quad \begin{array}{ll} \mathbf{return}\,() & \mapsto \mathrm{cooperate}\,(rs) \\ \langle \mathsf{yield}\,() \to s\rangle & \mapsto \mathrm{cooperate}\,(rs +\!\!+ [s]) \end{array} \end{array}$$

Example: lightweight threads (shallow handler)

Types

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

Handler

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

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

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

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

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

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

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

Example: lightweight threads (sheep handler)

Types

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

Handler

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

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

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

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

A1 B1 A2 B2

Part II

WebAssembly with effect handlers

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 $\langle$*typeidx*$\rangle$	define a new continuation type

Control tags

tag $\langle$*tagidx*$\rangle$	define a new tag

Core instructions

cont.new $\langle$*typeidx*$\rangle$	create a new continuation
suspend $\langle$*tagidx*$\rangle$	suspend the current continuation
resume (**tag** $\langle$*tagidx*$\rangle$ $\langle$*labelidx*$\rangle$)$*$	resume a continuation

Key ingredients

Continuation types

cont $\langle typeidx \rangle$	define a new continuation type

Control tags

tag $\langle tagidx \rangle$	define a new tag

Core instructions

cont.new $\langle typeidx \rangle$	create a new continuation
suspend $\langle tagidx \rangle$	suspend the current continuation
resume (**tag** $\langle tagidx \rangle$ $\langle labelidx \rangle$)$*$	resume a continuation

Additional instructions

cont.bind $\langle typeidx \rangle$	bind a continuation to (partial) arguments
resume_throw $\langle tagidx \rangle$	abort a continuation
barrier $\langle blocktype \rangle$ $\langle instr \rangle*$	block suspension

Control tags

Synonyms: operation, command, resumable exception, event

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

Continuations

Synonyms: stacklet, resumption

$\textbf{cont.new}\ \$ct : [(\textbf{ref}\ \$ft)] \to [(\textbf{ref}\ \$ct)]$	new continuation from function
where $\$ft$ denotes a function type $[s*] \to [t*]$	
$\$ct = \textbf{cont}\ \$ft$	
$\textbf{resume}\ (\textbf{tag}\ \$e\ \$l)* : [t1*\ (\textbf{ref}\ \$ct)] \to [t2*]$	resume 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 $(\textbf{tag}\ \$e\ \$l)*$: $[t1*\ (\textbf{ref}\ \$ct)] \to [t2*]$ — resume 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 $\$ct = \textbf{cont}\ ([t1*] \to [t2*])$
$\$exn : [s*] \to []$

Example: lightweight threads (application code)

$$
\begin{array}{ll}
(\textbf{type}\ \$\mathit{func}\ (\textbf{func})) & ;;\ [] \to [] \\
(\textbf{type}\ \$\mathit{cont}\ (\textbf{cont}\ \$\mathit{func})) & ;;\ \textbf{cont}\ ([] \to [])
\end{array}
$$

$$
\begin{array}{ll}
(\textbf{tag}\ \$\mathit{yield}) & ;;\ [] \to [] \\
(\textbf{tag}\ \$\mathit{fork}\ (\textbf{param}\ (\textbf{ref}\ \$\mathit{cont}))) & ;;\ [\textbf{cont}\ ([] \to [])] \to []
\end{array}
$$

Example: lightweight threads (application code)

```
(type $func (func))          ;; [] → []
(type $cont (cont $func))    ;; cont ([] → [])

(tag $yield)                          ;; [] → []
(tag $fork (param (ref $cont)))       ;; [cont ([] → [])] → []

(func $main
  (call $print (i32.const 0))
  (suspend $fork (cont.new (type $cont)
                           (ref.func $thread1)))
  (call $print (i32.const 1))
  (suspend $fork (cont.new (type $cont)
                           (ref.func $thread2)))
  (call $print (i32.const 2))
)

(func $thread1
  (call $print (i32.const 10))
  (suspend yield)
  (call $print (i32.const 11)))

(func $thread2
  (call $print (i32.const 20))
  (suspend yield)
  (call $print (i32.const 21)))
```

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 (handler code)

```
(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/WebAssembly/stack-switching/tree/main/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

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

Avoids code duplication

Barriers

Behaves like a catch-all handler that traps on suspension

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

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

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

Resuming with a unique prompt for the handler

$$\begin{array}{c}\textbf{resume\_with } (\textbf{tag } \$e\ \$l)* : [t1*\ (\textbf{ref } \$ct)] \rightarrow [t2*] \\ \text{where } \$ht = \textbf{handler } t2* \\ \$ct = \textbf{cont } ([(\textbf{ref } \$ht)\ t1*] \rightarrow [t2*])\end{array}$$

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

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

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 $\$\mathit{switch}$ by resuming to the continuation argument.

Direct switching

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

Switch directly to another continuation

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

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 $*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

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

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

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

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

Other extensions

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

Loser alternative: dynamically typed continuations

Features: continuations are mutable and no longer linear; each continuation is associated with a distinct prompt name; no tags

cont.new $\$ct : [(\textbf{ref } \$ft)] \to [(\textbf{ref } \$ct)]$	new continuation from function
suspend $\$ct : [t2{*}\ (\textbf{ref } \$ct)] \to [t1{*}]$	suspend continuation
resume $\$ct : [t1{*}\ (\textbf{ref } \$ct)] \to [t2{*}]$	resume continuation

where

$\$ft$ denotes a function type $[t1{*}] \to [t2{*}]$

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