Wasm with typed continuations continued

Sam Lindley

The University of Edinburgh

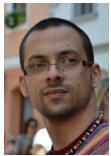
1st November 2021

Part I

Typed continuations

Wasm typed continuations proposal




...

Typed continuations to model stacks
https://github.com/WebAssembly/design/issues/1359

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

Formal spec
https://github.com/effect-handlers/wasm-spec/blob/master/proposals/continuations/Overview.md

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

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 ⟨*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, events

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

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

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

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

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

each $\$e$ is a control tag and

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

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

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

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

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))
```

Dependencies

Function references

Exceptions

Examples

Lightweight threads

Actors

Async/await

...

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

Partial continuation application

Analogous to **func**.**bind** in the function references proposal — but no need to do any allocation as continuations are one-shot

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

Partial continuation application

Analogous to **func**.**bind** in the function references proposal — but no need to do any allocation as continuations are one-shot

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

Avoids code duplication

Barriers

Behaves like a catch-all handler that traps on suspension

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

Part II

Extensions

Named handlers

Motivation: avoid linear dispatch

Named handlers

Motivation: avoid linear dispatch

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

handler $t*$

Named handlers

Motivation: avoid linear dispatch

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

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

Named handlers

Motivation: avoid linear dispatch

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

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

Direct switching

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

Switch directly to another continuation

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

Behaves as if we had a built-in tag

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

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

Direct switching

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

Switch directly to another continuation

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

Behaves as if we had a built-in tag

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

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

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

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)