



WebAssembly Spec Addendum: Legacy Exception Handling

WebAssembly Community Group
Andreas Rossberg (editor)

Feb 14, 2025

Contents

1	Introduction	1
2	Structure	1
3	Validation	2
4	Execution	4
5	Binary Format	7
6	Text Format	8
7	Index of Instructions	8

1 Introduction

This document describes an extension of the official WebAssembly standard developed by its [W3C Community Group](https://www.w3.org/community/webassembly/)¹ with additional instructions for exception handling. These instructions were never standardized and are deprecated, but they may still be available in some engines, especially in web browsers.

2 Structure

2.1 Instructions

¹ <https://www.w3.org/community/webassembly/>

Control Instructions

The set of recognised instructions is extended with the following:

```
instr ::= ...
      | try blocktype instr* (catch tagidx instr*)* (catch_all instr*)? end
      | try blocktype instr* delegate labelidx
      | rethrow labelidx
```

The instructions `try` and `rethrow`, are concerned with exceptions. The `try` instruction installs an exception handler, and may either handle exceptions in the case of `catch` and `catch_all`, or rethrow them in an outer block in the case of `delegate`.

The `rethrow` instruction is only allowed inside a `catch` or `catch_all` clause and allows rethrowing the caught exception by lexically referring to a the corresponding `try`.

When `try-delegate` handles an exception, it also behaves similar to a forward jump, effectively rethrowing the caught exception right before the matching `end`.

3 Validation

3.1 Conventions

Contexts

The context is enriched with an additional flag on label types:

```
labeltype ::= catch? resulttype
C          ::= {..., labels labeltype*, ...}
```

Existing typing rules are adjusted as follows:

- All rules that extend the context with new labels use an absent `catch` flag.
- All rules that inspect the context for a label ignore the presence of a `catch` flag.

Note

This flag is used to distinguish labels bound by `catch` clauses, which can be targeted by `rethrow`.

3.2 Instructions

Control Instructions

`try blocktype instr1* (catch x instr2*)* (catch_all instr3*)? end`

- The block type must be valid as some function type $[t_1^*] \rightarrow [t_2^*]$.
- Let C' be the same [context](#) as C , but with the [label type](#) $[t_2^*]$ prepended to the labels vector.
- Under context C' , the instruction sequence $instr_1^*$ must be valid with type $[t_1^*] \rightarrow [t_2^*]$.
- Let C'' be the same [context](#) as C , but with the [label type](#) `catch` $[t_2^*]$ prepended to the labels vector.
- For every x_i and $instr_{2i}^*$ in $(catch x instr_2^*)^*$:
 - The tag $C.tags[x_i]$ must be defined in the context C .
 - Let $[t_{3i}^*] \rightarrow [t_{4i}^*]$ be the tag type $C.tags[x_i]$.
 - The result type $[t_{4i}^*]$ must be empty.
 - Under context C'' , the instruction sequence $instr_{2i}^*$ must be valid with type $[t_{3i}^*] \rightarrow [t_2^*]$.
- If $(catch_all instr_3^*)^?$ is not empty, then:

- Under context C'' , the instruction sequence $instr_3^*$ must be valid with type $[] \rightarrow [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{\begin{array}{c} C \vdash blocktype : [t_1^*] \rightarrow [t_2^*] \quad C, labels [t_2^*] \vdash instr_1^* : [t_1^*] \rightarrow [t_2^*] \\ (C.tags[x] = [t^*] \rightarrow [])^* \\ C, labels (catch [t_2^*]) \vdash instr_2^* : [t^*] \rightarrow [t_2^*]^* \\ (C, labels (catch [t_2^*]) \vdash instr_3^* : [] \rightarrow [t_2^*])^? \end{array}}{C \vdash try blocktype instr_1^* (catch x instr_2^*)^* (catch_all instr_3^*)^? end : [t_1^*] \rightarrow [t_2^*]}$$

Note

The notation $C, labels (catch [t^*])$ inserts the new label type at index 0, shifting all others.

try blocktype instr delegate l*

- The label $C.labels[l]$ must be defined in the context.
- The block type must be valid as some function type $[t_1^*] \rightarrow [t_2^*]$.
- Let C' be the same **context** as C , but with the result type $[t_2^*]$ prepended to the labels vector.
- Under context C' , the instruction sequence $instr^*$ must be valid with type $[t_1^*] \rightarrow [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{C \vdash blocktype : [t_1^*] \rightarrow [t_2^*] \quad C, labels [t_2^*] \vdash instr^* : [t_1^*] \rightarrow [t_2^*] \quad C.labels[l] = [t_0^*]}{C \vdash try blocktype instr^* delegate l : [t_1^*] \rightarrow [t_2^*]}$$

Note

The label index space in the **context** C contains the most recent label first, so that $C.labels[l]$ performs a relative lookup as expected.

rethrow l

- The label $C.labels[l]$ must be defined in the context.
- Let $(catch^? [t^*])$ be the **label type** $C.labels[l]$.
- The catch must be present in the **label type** $C.labels[l]$.
- Then the instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$, for any sequences of value types t_1^* and t_2^* .

$$\frac{C.labels[l] = catch [t^*]}{C \vdash rethrow l : [t_1^*] \rightarrow [t_2^*]}$$

Note

The rethrow instruction is stack-polymorphic.

4 Execution

4.1 Runtime Structure

Stack

Exception Handlers

Legacy exception handlers are installed by try instructions. Instead of branch labels, their catch clauses have instruction blocks associated with them. Furthermore, a delegate handler is associated with a label index to implicitly rethrow to:

$$\begin{array}{lcl} \text{catch} & ::= & \dots \\ & | & \text{catch } \text{tagidx } \text{instr}^* \\ & | & \text{catch_all } \text{tagidx } \text{instr}^* \\ & | & \text{delegate } \text{labelidx} \end{array}$$

Administrative Instructions

Administrative instructions are extended with the caught instruction that models exceptions caught by legacy exception handlers.

$$\begin{array}{lcl} \text{instr} & ::= & \dots \\ & | & \text{caught}_n \{ \text{exnaddr} \} \text{instr}^* \text{ end} \end{array}$$

Block Contexts

Block contexts are extended to include caught instructions:

$$\begin{array}{lcl} B^k & ::= & \dots \\ & | & \text{caught}_n \{ \text{exnaddr} \} B^k \text{ end} \end{array}$$

Throw Contexts

Throw contexts are also extended to include caught instructions:

$$\begin{array}{lcl} T & ::= & \dots \\ & | & \text{caught}_n \{ \text{exnaddr} \} T \text{ end} \end{array}$$

4.2 Instructions

Control Instructions

try *blocktype* *instr*₁^{*} (catch *x* *instr*₂^{*})^{*} (catch_all *instr*₃^{*})[?] end

1. Assert: due to validation, $\text{expand}_F(\text{blocktype})$ is defined.
2. Let $[t_1^m] \rightarrow [t_2^n]$ be the function type $\text{expand}_F(\text{blocktype})$.
3. Let L be the label whose arity is n and whose continuation is the end of the try instruction.
4. Assert: due to [validation](#), there are at least m values on the top of the stack.
5. Pop the values val^m from the stack.
6. Let F be the current frame.
7. For each catch clause (catch x_i *instr*_{2*i*}^{*}) do:
 - a. Assert: due to [validation](#), $F.\text{module.tagaddrs}[x_i]$ exists.
 - b. Let a_i be the tag address $F.\text{module.tagaddrs}[x_i]$.
 - c. Let catch_i be the catch clause (catch a_i *instr*_{2*i*}^{*}).

8. If there is a catch-all clause ($\text{catch_all } instr_3^*$), then:
 - a. Let $catch'^{?}$ be the handler ($\text{catch_all } instr_3^*$).
9. Else:
 - a. Let $catch'^{?}$ be empty.
10. Let $catch^*$ be the concatenation of $catch_i$ and $catch'^{?}$.
11. Enter the block $val^m instr_1^*$ with label L and exception handler $\text{handler}_n\{catch^*\}^*$.

$$\begin{aligned}
 F; val^m (\text{try } bt \text{ } instr_1^* (\text{catch } x \text{ } instr_2^*)^* (\text{catch_all } instr_3^*)^? \text{ end} \hookrightarrow \\
 F; \text{label}_n\{\epsilon\} (\text{handler}_n\{(\text{catch } a_x \text{ } instr_2^*)^* (\text{catch_all } instr_3^*)^?\} val^m instr_1^* \text{ end}) \text{ end} \\
 (\text{if } \text{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n] \wedge (F.\text{module.tagaddrs}[x] = a_x)^*)
 \end{aligned}$$

try blocktype instr* delegate l

1. Assert: due to validation, $\text{expand}_F(\text{blocktype})$ is defined.
2. Let $[t_1^m] \rightarrow [t_2^n]$ be the function type $\text{expand}_F(\text{blocktype})$.
3. Let L be the label whose arity is n and whose continuation is the end of the try instruction.
4. Let H be the **exception handler** l , targeting the l -th surrounding block.
5. Assert: due to **validation**, there are at least m values on the top of the stack.
6. Pop the values val^m from the stack.
7. Enter the block $val^m instr^*$ with label L and exception handler $\text{HANDLER}_n\{\text{DELEGATE-}l\}$.

$$\begin{aligned}
 F; val^m (\text{try } bt \text{ } instr^* \text{ delegate } l) \hookrightarrow F; \text{label}_n\{\epsilon\} (\text{handler}_n\{\text{delegate } l\} val^m instr^* \text{ end}) \text{ end} \\
 (\text{if } \text{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n])
 \end{aligned}$$

throw_ref

1. Let F be the current frame.
2. Assert: due to validation, a reference is on the top of the stack.
3. Pop the reference ref from the stack.
4. If ref is $\text{ref.null } ht$, then:
 - a. Trap.
5. Assert: due to validation, ref is an exception reference.
6. Let $\text{ref.exn } ea$ be ref .
7. Assert: due to validation, $S.\text{exns}[ea]$ exists.
8. Let exn be the exception instance $S.\text{exns}[ea]$.
9. Let a be the tag address $exn.\text{tag}$.
10. While the stack is not empty and the top of the stack is not an **exception handler**, do:
 - a. Pop the top element from the stack.
11. Assert: the stack is now either empty, or there is an exception handler on the top of the stack.
12. If the stack is empty, then:
 - a. Return the exception ($\text{ref.exn } a$) as a result.
13. Assert: there is an **exception handler** on the top of the stack.
14. Pop the exception handler $\text{handler}_n\{catch^*\}$ from the stack.

15. If $catch^*$ is empty, then:
 - a. Push the exception reference $ref.exn\ ea$ back to the stack.
 - b. Execute the instruction $throw_ref$ again.
16. Else:
 - a. Let $catch_1$ be the first catch clause in $catch^*$ and $catch'^*$ the remaining clauses.
 - b. If $catch_1$ is of the form $catch\ x\ l$ and the exception address a equals $F.module.tagaddrs[x]$, then:
 - i. Push the values $exn.fields$ to the stack.
 - ii. Execute the instruction $br\ l$.
 - c. Else if $catch_1$ is of the form $catch_ref\ x\ l$ and the exception address a equals $F.module.tagaddrs[x]$, then:
 - i. Push the values $exn.fields$ to the stack.
 - ii. Push the exception reference $ref.exn\ ea$ to the stack.
 - iii. Execute the instruction $br\ l$.
 - d. Else if $catch_1$ is of the form $catch_all\ l$, then:
 - i. Execute the instruction $br\ l$.
 - e. Else if $catch_1$ is of the form $catch_all_ref\ l$, then:
 - i. Push the exception reference $ref.exn\ ea$ to the stack.
 - ii. Execute the instruction $br\ l$.
 - f. Else if $catch_1$ is of the form $catch\ x\ instr^*$ and the exception address a equals $F.module.tagaddrs[x]$, then:
 - i. Push the caught exception $caught_n\{ea\}$ to the stack.
 - ii. Push the values $exn.fields$ to the stack.
 - iii. Enter the catch block $instr^*$.
 - g. Else if $catch_1$ is of the form $catch_all\ instr^*$, then:
 - i. Push the caught exception $caught_n\{ea\}$ to the stack.
 - ii. Enter the catch block $instr^*$.
 - h. Else if $catch_1$ is of the form $delegate\ l$, then:
 - i. Assert: due to validation, the stack contains at least l labels.
 - ii. Repeat l times:
 - While the top of the stack is not a label, do:
 - Pop the top element from the stack.
 - iii. Assert: due to validation, the top of the stack now is a label.
 - iv. Pop the label from the stack.
 - v. Push the exception reference $ref.exn\ ea$ back to the stack.
 - vi. Execute the instruction $throw_ref$ again.
 - i. Else:
 1. Push the modified handler $handler_n\{catch'^*\}$ back to the stack.
 2. Push the exception reference $ref.exn\ ea$ back to the stack.
 3. Execute the instruction $throw_ref$ again.

$$\begin{array}{ll}
\text{handler}_n\{(\text{catch } x \text{ instr}^*) \text{ catch}^*\} T[(\text{ref.exn } a) \text{ throw_ref}] \text{ end} & \hookrightarrow \text{caught}_n\{a\} \text{ exn.fields instr}^* \text{ end} \\
& (\text{if } \text{exn} = S.\text{exns}[a] \\
& \quad \wedge \text{exn.tag} = F.\text{module.tagaddrs}[x]) \\
\text{handler}_n\{(\text{catch_all instr}^*) \text{ catch}^*\} T[(\text{ref.exn } a) \text{ throw_ref}] \text{ end} & \hookrightarrow \text{caught}_n\{a\} \text{ instr}^* \text{ end} \\
B^l[\text{handler}_n\{(\text{delegate } l) \text{ catch}^*\} T[(\text{ref.exn } a) \text{ throw_ref}] \text{ end}] & \hookrightarrow (\text{ref.exn } a) \text{ throw_ref}
\end{array}$$

rethrow l

1. Assert: due to **validation**, the stack contains at least $l + 1$ labels.
2. Let L be the l -th label appearing on the stack, starting from the top and counting from zero.
3. Assert: due to **validation**, L is a catch label, i.e., a label of the form (catch $[t^*]$), which is a label followed by a caught exception in an active catch clause.
4. Let a be the caught exception address.
5. Push the value `ref.exn a` onto the stack.
6. Execute the instruction `throw_ref`.

$$\text{caught}_n\{a\} B^l[\text{rethrow } l] \text{ end} \quad \hookrightarrow \quad \text{caught}_n\{a\} B^l[(\text{ref.exn } a) \text{ throw_ref}] \text{ end}$$

Entering a catch block

1. Jump to the start of the instruction sequence $instr^*$.

Exiting a catch block

When the end of a catch block is reached without a jump, thrown exception, or trap, then the following steps are performed.

1. Let val^m be the values on the top of the stack.
2. Pop the values val^m from the stack.
3. Assert: due to validation, a caught exception is now on the top of the stack.
4. Pop the caught exception from the stack.
5. Push val^m back to the stack.
6. Jump to the position after the end of the administrative instruction associated with the caught exception.

$$\text{caught}_n\{a\} \text{ val}^m \text{ end} \hookrightarrow \text{val}^m$$

Note

A caught exception can only be rethrown from the scope of the administrative instruction associated with it, i.e., from the scope of the `catch` or `catch_all` block of a legacy `try` instruction. Upon exit from that block, the caught exception is discarded.

5 Binary Format

5.1 Instructions

Control Instructions

```

instr ::= ...
| 0x06 bt:blocktype (in1:instr)*
  (0x07 x:tagidx (in2:instr)*)*
  (0x19 (in3:instr)*)? 0x0B      ⇒ try bt in1* (catch x in2*)* (catch_all in3*)? end
| 0x06 bt:blocktype (in:instr)*
  0x18 l:labelidx                ⇒ try bt in* delegate l
| 0x09 l:labelidx                ⇒ rethrow l

```

6 Text Format

6.1 Instructions

Control Instructions

The label identifier on a structured control instruction may optionally be repeated after the corresponding end, else, catch, catch_all, and delegate pseudo instructions, to indicate the matching delimiters.

```

blockinstrI ::= ...
| 'try' I':labelI bt:blocktype (in1:instrI')*
  ('catch' id1? x:tagidxI (in2:instrI')*)*
  ('catch_all' id1? (in3:instrI')*)*
  'end' id2?
  ⇒ try bt in1* (catch x in2*)* (catch_all in3*)? end
  (if id1? = ε ∨ id1? = label, id2? = ε ∨ id2? = label)
| 'try' I':labelI bt:blocktype (in1:instrI')*
  'delegate' l:labelidxI l:labelidxI
  ⇒ try bt in1* delegate l (if id? = ε ∨ id? = label)

plaininstrI ::= ...
| 'rethrow' l:labelidxI ⇒ rethrow l

```

7 Index of Instructions

Instruction	Binary Opcode	Type	Validation	Execution
try <i>bt</i>	0x06	$[t_1^*] \rightarrow [t_2^*]$	validation, validation	execution, execution
catch <i>x</i>	0x07		validation	execution
rethrow <i>n</i>	0x09	$[t_1^*] \rightarrow [t_2^*]$	validation	execution
delegate <i>l</i>	0x18		validation	execution
catch_all	0x19		validation	execution