WebAssembly Exception Handling (Phase 1)

Heejin Ahn and Michael Starzinger

Involved parties

- Champion / WebAssembly Toolchain Implementation: Heejin Ahn
- Google WebAssembly Toolchain TLM: Derek Schuff
- Google WebAssembly V8 TL / V8 Implementation: Michael Starzinger

Agenda

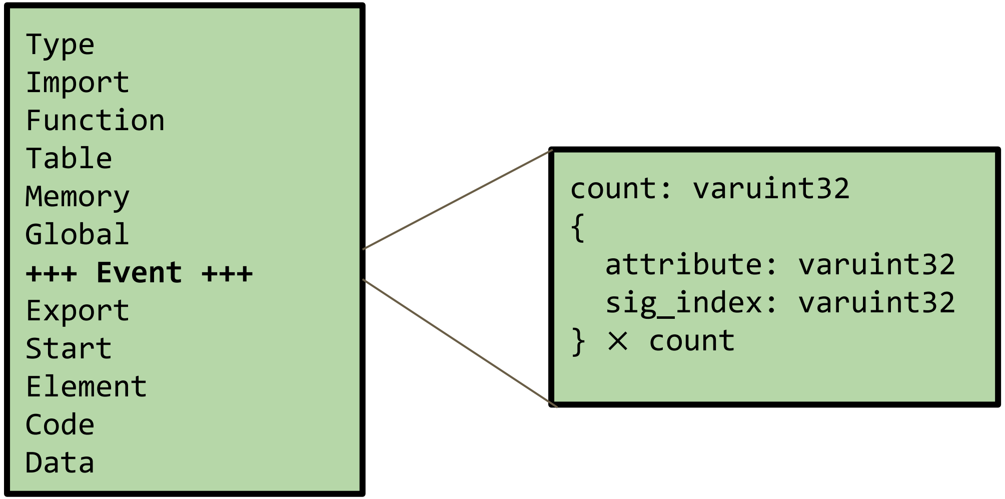
- Proposed goals
- Spec recap
- Status updates
- Discussion: Should we catch traps with `catch`?

Proposed goals

- Provide a primitive for exception handling for Wasm programs
    - Zero-cost: no runtime cost until exceptions are thrown or caught
    - Structured: composes properly with existing control flow constructs
    - Safe: fast, single-pass verification
- Low-cost C++ exception handling
    - Emscripten currently indirects through JS for try/catch (expensive, not production quality)
    - WASM solution should be nearly as efficient as a native target

Event section

- Wasm events are features that suspend the current execution and transfer the control flow to a corresponding handle
    - Only supported kind is exceptions now



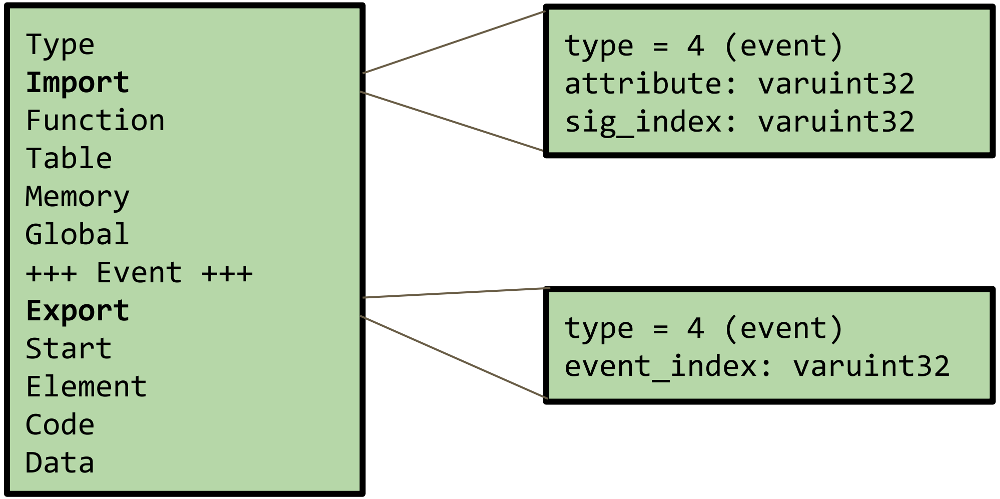
```
Type
Import
Function
Table
Memory
Global
+++ Event +++
Export
Start
Element
Code
Data
```

```
count: varuint32
{
  attribute: varuint32
  sig_index: varuint32
} × count
```

- Event section declares a list of event types
- `attribute`: application-specified number (0 for exceptions)
- `sig_index`: index into types section of function signature

Event import/export

- Events can be imported and exported from a module
- Each instantiation of a module produces new event tags



Type
Import
Function
Table
Memory
Global
+++ Event +++
Export
Start
Element
Code
Data

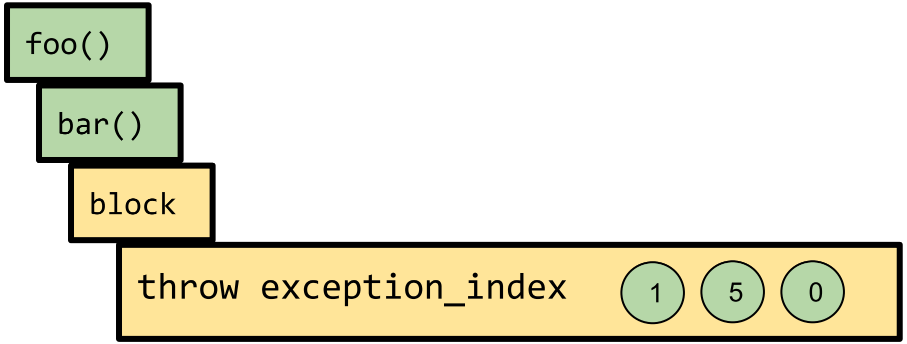
```
type = 4 (event)
attribute: varuint32
sig_index: varuint32
```

```
type = 4 (event)
event_index: varuint32
```

- Imported events specify an attribute and an expected signature
- Exported events specify an exported event index

Throwing an exception

- A **`throw`** accepts its arguments on the stack and creates an **`except_ref`** value out of them
- and begins searching the *control* and *call* stacks for a handler



Google

The exception reference data type

- EH proposal requires the reference types proposal as a prerequisite
- `except_ref` type is a subtype of `anyref`
- `except_ref` type contains values thrown
- `except_ref` type possibly can contain more information, such as a stack trace

try and **catch** blocks

- **try** … **catch** … **end** introduces a new block kind
  - **try** can have a label for branches too
- If any instruction between a **try** and a **catch** throws, the VM unwinds the wasm execution stack and resumes execution from the **catch**
- A **catch** pushes an **except_ref** value onto the stack

```
try [block_type]
  ...
catch
  ...
end
```

Rethrowing an exception

- A `rethrow` takes an **except_ref** value from the stack (produced by a `catch`) and continue unwinding the execution stack with the exception
- A `rethrow` can occur anywhere (not necessarily between `catch` and `end`)

Exception data extraction

- A **`br_on_exn`** checks the exception tag of an **`except_ref`** on top of the stack (without popping it) if it matches the given exception index
    - If they match, it
        - branches out to the label referenced
        - pops the **`except_ref`** from the stack
        - extracts the **`except_ref`** and pushes the exception's values onto the stack
    - If they don't match, it does nothing, and the **`except_ref`** remains on the stack
- Format: `br_on_exn label except_index`

Exception data extraction

- We can also test an **`except_ref`** against multiple tags

```
block $l0 (result i32, i64)
  block $l1 (result i32)
    ...
    ;; except_ref $e is on the stack at this point
    br_on_exn $l1 e(i32)      ;; branch to $l1 with $e's arguments
    ;; except_ref $e is left on the stack if br_on_exn is not taken
    br_on_exn $l0 e(i32, i64) ;; branch to $l0 with $e's arguments
    rethrow
  end
  ;; handler for $l1
end
;; handler for $l0
```

Toolchain implementation status

- Current status
  - LLVM part is mostly done, modulo proper `throw;` keyword support
  - Other toolchain support in progress (Binaryen / emscripten)
- Next steps: short term
  - Finish toolchain and libcxxabi support
  - Start end-to-end testing
- Next steps: long term
  - Support `throw;` keyword properly
    - Requires full reference type proposal support, including exception tables to maintain a stack of in-flight `except_refs`
  - setjmp-longjmp handling
  - Benchmark tests and optimizations

V8 Implementation Status (1)

- Current status
  - Full support of current proposal in top-tier and interpreter (for debugging)
  - Behind the `--experimental-wasm-eh` runtime flag
- Interaction with other language features
  - Support for **`except_ref`** in signatures, locals, and globals, and (partially) tables.
- Implementation details
  - **`try`** … **`catch`** … **`end`** implemented as “zero-cost” for non-exceptional execution
  - Exception packages (**`except_ref`** values) allocated on garbage collected heap (**`throw`** allocates)

V8 Implementation Status (2)

- Interaction with JavaScript embedding
  - Exported exceptions/events are an instance of `WebAssembly.Exception`
  - Exception packages are instances of `WebAssembly.RuntimeError`
  - Exception package contain full stacktrace (WebAssembly & JavaScript frames)
  - No access to exception tags/values encoded in package from JavaScript
  - Cannot construct WebAssembly exceptions from JavaScript (neither package nor event)
  - All JavaScript exceptions caught by **`catch`**, can be **`rethrow`**n but never match **`br_on_exn`**
  - In JavaScript **`anyref`** and **`except_ref`** are indistinguishable (can be any JavaScript value)
- Current interaction with traps (self-consistent but undesirable)
  - **`try`** … **`catch`** … **`end`** will catch traps only if it crosses one or more function invocations
  - Conceptually: traps are converted into exceptions at function boundary

Should a `catch` catch traps?

```
try
  local.get 0
  local.get 1
  i32.div_s     ;; possibly divide by zero
  ...
catch
  ;; recover from the trap??
end
```

What kind of traps do we have?

- Undefined arithmetics (e.g. divide by zero)
- Invalid memory/table accesses
- Ill-typed indirect function calls
- GC-related traps

Reasons for catching traps?

- Error recovery and continuing execution / Backtrace preservation
    - Which traps can be recovered from and how?
- Invariant: All non-local control-flow transfers can be intercepted by catch
    - Otherwise traps would represent a non-local control-flow transfer from WebAssembly to the embedder (e.g. JavaScript) that is impossible to notice/intercept by a surrounding function when functions/modules are composed

Caveats

- Catching traps in wasm does NOT mean languages' `catch` clauses do too. Then how useful would catching traps be?
    - Some languages' (e.g. C++) `catch` clauses don't catch traps
        - In case someone wants to do that, it's hard
    - Some languages (e.g. Rust or Java) turn trap-like behaviors to exceptions, so wouldn't be useful for those
        - Compiler-generated code include checks for trappable instructions
- We need extra checking code after every catch instruction
    - *"Is this a trap? Then don't run destructors / cleanup code and just rethrow"* per every catch block, including all landing pads created by destructor frames

Representation of traps

- Currently in browsers wasm traps surface as JS exceptions, from which point they become ‘foreign exceptions’ which are caught
    - Whether we catch traps or not, we should at least be consistent before we reach embedder and after
- We need a way to distinguish traps from foreign exceptions anyway
    - Special trap tag?

Discussions

Google

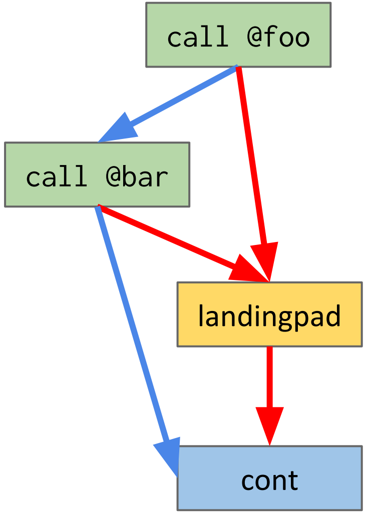
Backup Slides

Google

Want to make C++'s **catch** clause catch traps?

- Every throwable call ends a BB and has a normal exit and an unwind exit, but other trappable instructions don't
- After CFG transformation, we don't guarantee traps end up in the correct **catch** clause

```
try {
  foo();
  bar();
} catch (...) {
}
```



Want to make C++'s **`catch`** clause catch traps?

1. High-overhead solutions: Not zero-cost anymore
    - Solution a: Insert unwind edges after all other trappable instructions
    - Solution b: Outline every region between a `try` and a `catch` clause
        - Windows SEH takes this approach
2. Compiler inserts code so that after **`catch`** instruction, if it's a trap, always rethrow it until we reach the top frame
    - Involves small code size overhead of checking and rethrowing after every **`catch`** instruction
    - How is it different from not catching traps?