WEBPACK + WEBASSEMBLY

WEBPACK AND THE CHALLENGE OF WEBASSEMBLY

INTRODUCTION
WEBASSEMBLY AND ESM

WebAssembly?

- Low-Level Binary Format for Code
- Typed (i8 – i64, f32, f64)
- Memory
- Usually compiled from native languages (C/C++, Rust, …)
- Assembly-like Textual Representation (WAT)
- WASM = WebAssembly

Structure of a WASM-file

- Multiple Sections:
  - Import
  - Globals
  - Function-types
  - Code
  - Data
  - Export
  - Start
  - Other and also Custom sections

EXAMPLE WAT FILE

```
(module
    (type $addType (func (param i32) (result i32)))
    (type $getNumberType (func (result i32)))
    (import "./module" "getValue" (func $getValue (type $getNumber_type)))
    (func $add (export "add") (type $add_type) (param $p0 i32) (result i32)
        (i32.add
            (call $getValue)
            (get_local $p1)
        )
    )
    (func $getNumber (export "getNumber") (type $getNumber_type) (result i32)
        (i32.const 40)
    )
)
```

WebAssembly API

- 1. fetch the binary
- 2. compile a `WebAssembly.Module`
- 3. instanciate a `WebAssembly.Instance`
  - An `importsObject` can be passed
- 4. Use `instance.exports`

- 2. and 3. are available in two variants: Async and Sync

ECMASCRIPT MODULES?

- (New) source type "module" next to "script"
- Allows new Syntax:
  - Import
  - Export
- Always strict
- ESM = EcmaScript Module

EXPORTS, IMPORTS AND LIVE-BINDINGS

- `export { x, y };`
  - Does not export an object
  - Does not export the values of `x` and `y`
  - It exports bindings to the variables `x` and `y` in this scope!
- `import { x, y } from "module";`
  - Binds `x` and `y` in the new scope (read-only)
  - Any change in the origin module is reflected here

ESM LIFECYCLE

- Phase 1: Recursively:
  - Load and Parse files
  - Resolve Imports
- Phase 2: "Connect" / Bind exports and imports
- Phase 3: Evaluate all code at once (in the same microtask/tick)
  - Dependencies are evaluated before Parent (except in circles)
  - Dependencies are evaluated in order of Occurrence (imports)

ADDING WASM TO WEBPACK

The Story

- So we want to add WebAssembly support to webpack.
- How to do that?
  - Let's start with an prototype and find problems
  - Iterate on that until all Goals and Milestones are completed

Mozilla sponsorship

- We got a sponsorship from Mozilla to work on that
  - Seems like they want to push WASM and Rust forward
- This can cover a part of the implementation cost

GOALS

- EASY TO USE
- HIDE TECHNICAL DETAILS
- WEBPACK DOESN'T CARE ABOUT HIGH-LEVEL LANGUAGE (C/C++, RUST)
- PERFORMANT (PRIMARY RUNTIME, BUT ALSO BUILD)
- WASM FILE IS A MODULE
- INTEGRATE WELL WITH ESM
- CODE SPLITTING SUPPORT

GOAL AS SYNTAX

- `import { func } from "./module.wasm";`
- `import { fn1 } from "./module.wat";`
- `import { fn2 } from "./module.rs";`
- `import { fn3 } from "./module.cpp";`

- `(import "./memory.wasm" "memory" (memory $mem 1))`

IDEA

- Add WASM as module type
- Run async operations on Split Points (i. e. `import()`)
- Treat import and exports section like in ESM
- Enforce single instance similar to ESM
- Run Start section on "Evaluate"

Current State (webpack 3)

- webpack only support a single module type: JavaScript (JS)
  - Everything is compiled to JS
- webpack doesn't support additional operations on Split Points

WEBPACK CHANGE PLAN

- ADD SUPPORT FOR MULTIPLE MODULE TYPES
  - JS/AUTO
  - JS/ESM
  - WASM
  - JSON
  - CSS (SEPARATE PLUGIN)
- ADD SUPPORT FOR EMITTING MULTIPLE ASSETS PER CHUNK + RUNTIME CODE

Challenge 1: ESM spec vs WebAssembly API

ECMAScript Module

- Life-cycle
  - 1. Load
  - 2. Parse
  - 3. Bind Exports and Imports
  - 4. Evaluate (single Microtask)

WebAssembly Module

- Wasm API
  - 1. fetch
  - 2. compile
  - 3. Provide imports
  - 4. instantiate, Evaluate Start
  - 5. Receive exports

WebPack Change Plan

- Add Parser for WebAssembly to extract import and export section
- Use imports in section as dependencies
- Runtime code: fetch + compile at the Split Point
- Runtime code:
  - load dependencies and create `importObject`
  - instantiate sync
  - setup exports

SIMPLIFIED GENERATED CODE

```
// at split point
res = await fetch(url);
wasmModule = await WebAssembly.compileStreaming(res);

// at evaluate
instance = new WebAssembly.Instance(wasmModule, {
  importedModule: __webpack_require__(123)
});
module.exports = instance.exports;
```

Problems

- Chrome throws Error when using sync `instantiate` with binary > 4kb
  - This is against the Spec!
- Safari doesn't compile in "`compile`", but in `instantiate`
  - For "technical" reasons...

- We MUST use `instantiateStreaming` instead of `compileStreaming` + sync `instantiate`!

Challenge 2: async instantiate

- We need to move `instantiate` to the Split Point
- This means `instantiate` could be called before dependencies are evaluated
  - Start section would run in wrong Phase
  - Import handles values and not bindings
    - Values are copied / snapshot on `instantiate`
    - Values are not available before dependencies are evaluated
- It took a bit until we had a "solution" for this problem…

New plan

- Rewrite the WASM binary
  - Remove Start section
  - Rewrite imported Globals to mutable Globals
  - Create a new exported `_webpack_init_` function
    - Set imported globals (passed as arguments)
    - Call old Start function
- Imported functions get wrapped in trampoline function
  - Even enabled live-bindings for functions

REWRITING WASM

```
(func $something
  (import "./module" “something")
  (param i32))

(global $magicNumber
  (import "./a.js" "magicNumber")
  i32)

(func $start
  get_global $magicNumber
  call $something
)

(start $start)
```

```
(func $something
  (import "./module" “something")
  (param i32))

(global $magicNumber (mut i32) (i32.const 66))

(func $start
  get_global $magicNumber
  call $something
)

(func $__webpack_init__
  (export "__webpack_init__") (param $p0 i32)
  get_local $p0
  set_global $g0
  call $start
)
```

INSTANTIATE AT SPLIT POINT & TRAMPOLINE

```
// at split point
instance = new WebAssembly.Instance(wasmModule, { "./module": {
  "something": function(p0i32) {
    return installedModules[123].exports["something"](p0i32);
  }
} });

// at evaluate
module.exports = instance.exports;
__webpack_require__(123);
a = __webpack_require__(321);
instance.exports.__webpack_init__(
  a.magicNumber
)
```

Challenge 3: Memory and Tables

- Importing Memory and Tables doesn't work
  - Can't be passed as arguments to the init function
  - Can't use trampolines (no functions)
  - MUST be provided on instantiate

Extra plan

- Forbit importing Memory or Table from non-WASM into WASM
- For Wasm-to-Wasm imports:
  - Handle Memory and Table directly in runtime code
  - May sequentialize instantiate (Download and Compile is still parallel)

Challenge 4: Reexported Globals and Init

- Mutable Globals can't be exported (at this time)
  - Problem when exporting and imported global
- One can provide an expression for initializing the global
  - This can be "get other global"
  - Problem when other global is imported
  - Extra problem when this global is exported

```
;; importing a global
(import "module" "abc" (global i32))

;; re-exporting this global
(export "def" (global 0))

;; global with init expression
(global $g i32 (get_global 0))

;; export this global
(export "ghi" (global $g))
```

Extra plan (the second)

- Detect these special cases
  - May move init expression to our `_webpack_init_` function
  - Handle reexporting in the runtime code
    - Problematic exports are handled in JS instead of WASM

Challenge 5: i64

- WebAssembly has this nice 64bit integer data type
- JavaScript can't handle it (currently)*
- Any function with `i64` in signature will crash in JS (`RuntimeError`)
  - But it can be passed to other WebAssembly modules
  - Our Trampoline will also crash

* `BigInt` could

Extra plan (3rd)

- Handle these functions in runtime
  - Enforce direct connecting these modules
  - Handled like Memory and Table types
- Emit a build warning when importing these i64-functions into ESM
  - Can't always detect it: Will `RuntimeError` anyway

IT WORKS

- Unsolved Limitations:
  - Can't import memory/table from JS
  - Waterfall-like instantiate when direct connection are required
  - Trampoline functions have extra overhead
    - We try to omit them when possible
- So far these limitations have not become problems
  - All compile-to-wasm languages embed and export the memory
  - Multiple WebAssembly modules in a single chunk are rare (currently)

The Spec

- There is now a spec proposal for WASM-ESM-integration
  - Non-function ESM-to-WASM imports are disallowed
    - Yeah, our limitation is now baked by the spec
    - We can remove special handling for globals
  - Mostly compatible with our implementation
    - Exported Globals appear as `WebAssembly.Global` in JS
  - Next step: Full compatibility with the spec

Recent developments

- Spec proposal merged: Importing and Exporting mutable globals
  - Changed global appearance in JS
  - WASM-to-WASM globals need to be directly connected

Optimizations

- "Mangle" export/import and module names in binary
  - Saves a few bytes
    - Becomes more relevant when using many small modules
- Tree Shaking WASM exports
  - We remove the exports, but have no DCE
  - We need a WASM minimizer!

The future

- Provide or generate fallback JS when WASM fails to load
  - `import("./module").catch(e => import("./fallback"))`
  - Do we need a transparent solution? I. e. `.wasm.js` as fallback
- `SharedMemoryBuffer` + WASM threads
  - How to pass the memory? How to instantiate in worker? Spec doesn't cover.
- Reference types, Garbage Collection and structured objects
  - We may need to handle this on boundaries

THE FUTURE

- `BigInt` FOR `i64`
- CODE SPLITTING CONSTRUCT FOR WASM
  - JS HAS `import("module")`. DO WE NEED A CONSTRUCT FOR WASM?

MY OPINION

- MANY SMALL WASMS > ONE BIG WASM
  - SIMILAR TO JS, CODE SPLITTING, TREE SHAKING
- HIDING ASYNC IS IMPORTANT FOR TRANSPARENT USAGE IN LIBRARIES
- CODE SPLITTING STAYS IMPORTANT WITH WASM

THE END

https://github.com/sokra/slides