JS API for suspending wasm on promises

ROSS TATE

IN COLLABORATION WITH

FRANCIS MCCABE

LUKE WAGNER

ALON ZAKAI

Two-Pronged Plan

JS API for Async/Await

- No change to wasm
- Only changes JS API
- Solely supports interop with JS's async/await paradigm
- Ships first and soon

Wasm for First-Class Stacks

- Changes wasm
- No change to JS API???
- Supports language/runtime features utilizing stacks
- Ships second

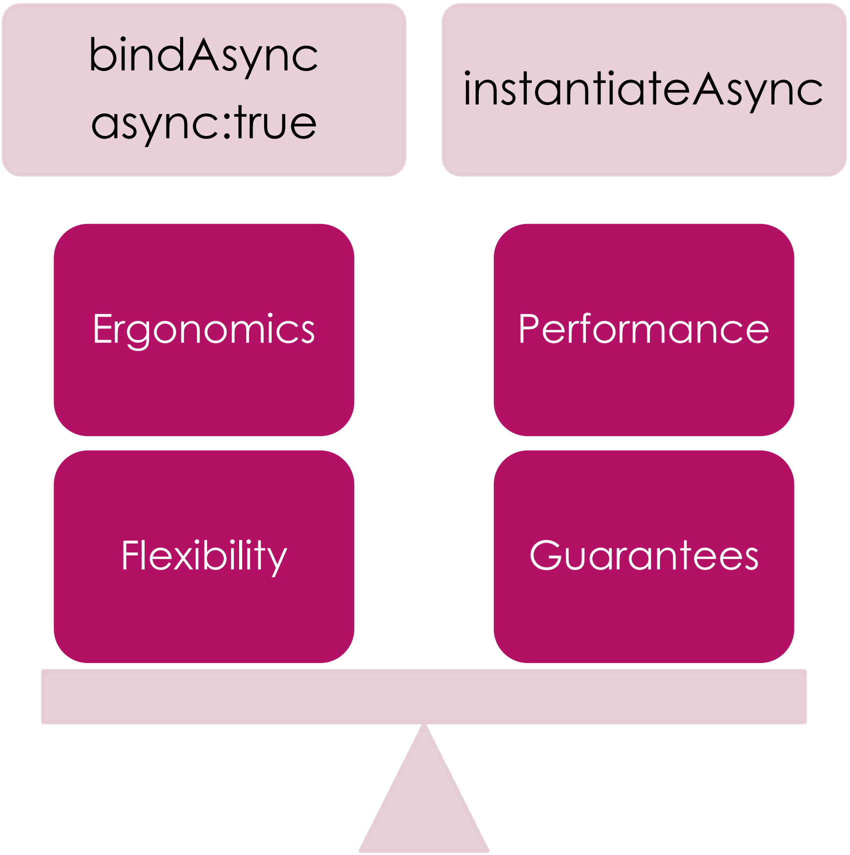
Considerations

AND RATIONALE

Understanding Usage Scenarios

- Two classes of exports
  - “asynchronous” exports like main, which call “asynchronous” imports
  - “shallow” exports, such as getters/setters/malloc, that call no (asynchronous) imports
- Imports
  - “asynchronous” imports call async JS functions or “asynchronous” exports of **other** wasm modules
  - call “shallow” exports of **same** module
- Forwards-compatibility
  - “asynchronous” imports that call “asynchronous” exports of **same** wasm module
  - “multi-suspension” exports – multiple promises live at a time for same wasm module

Balancing Tradeoffs



bindAsync
async:true

instantiateAsync

Ergonomics

Performance

Flexibility

Guarantees

🚫 Preserving Invariants

- Wasm programs often have a shadow stack
- (Properly implemented) shadow stack is guaranteed to be aligned with wasm stack
  - At least so far
- But suspending wasm stack does not suspend the shadow stack
  - An imported function can return without exports it called having returned
  - Can cause shadow stack to become misaligned
  - More generally violates program invariants that were previously guaranteed
- instantiateAsync prevented this by wrapping **all** exports and imports
  - But this was too restrictive, particularly for calling “shallow” getters/setter exports

🚫 JS-Wasm Interchangeability

- Many functions implementable in both wasm and JS
  - But only wasm functions will be suspendable
- Trapping on suspend with JS frame on stack introduces semantic difference
  - Even for functions that otherwise are identical in wasm vs. JS
- instantiateAsync prevented exposing semantic difference
  - Ensured every "suspension" event had a "matching" handler with no JS frames in between
- Too restrictive
  - No type information to distinguish "asynchronous" exports from "shallow" exports

🚫JS-wasm cross-call implementation

- Currently no stack switch when wasm calls JS
- If wasm can run on separate stack, then calls to JS might have to stack switch
- instantiateAsync prevented changes to JS calls by wrapping **all** imports
  - Non-async wasm still calls to JS as before
  - Async wasm's import wrapper performs the stack switch
- Too difficult to ensure
  - Requires way to ensure all imports are wrapped (wasm-ESM and funcrefs are particularly difficult)

☑ Implementation Interchangeability

- instantiateAsync just generates a wasm program that handles promises
- externref already enables wasm programs to do so
  - Continuation-passing-style compilation to wasm can easily handle promises
  - Wasm-to-wasm transpiler can add efficient promise-handling functionality
- instantiateAsync is completely interchangeable with other implementation strategies
  - Strong implementation abstraction
  - Requires every “suspension” event to have at most one applicable “matching” handler
  - Easy to ensure

High-Level Summary of Rationale

- instantiateAsync wraps whole instance
    - Problem: too restrictive – does not differentiate synchronous vs asynchronous imports/exports
- Solution: use Luke’s approach of modifying each import and export individually
    - New problem: connection between imports and exports is lost
        - Up to where should a modified import suspend to?
        - Using “most recent” is a leaky abstraction – loses implementation interchangeability due to accidental handling
- Solution: introduce construct to explicitly match modifications of imports and exports
    - Added benefit: easy to implement efficiently due to explicitness

Syntax

JS API

```
interface Suspender {
    constructor();
    Function suspendOnReturnedPromise(Function); // wraps imports
    Function returnPromiseOnSuspend(Function); // wraps exports
}
```

Example Usage

demo.wasm

```
(module
    (import "js" “syncimp" (func $si))
    (import "js" “asyncimp" (func $ai (result i32)))
    (func $init (call $si))
    (start $init)
    (func (export “main") (result i32) (call $ai))
)
```

demo.js

```
var suspender = new Suspender();
var importObj = {js: {
    syncimp: () => console.log("hello,"),
    asyncimp: suspender.suspendOnReturnedPromise(
        () => fetch('data.txt').then(res => res.text()).then(txt => parseFloat(txt)))
}};
fetch('demo.wasm').then(response => response.arrayBuffer()
).then(buffer => WebAssembly.instantiate(buffer, importObj)
).then(({module, instance}) => {
    var main = suspender.returnPromiseOnSuspend(instance.exports.main);
    return main();
}).then(num => ...);
```

Semantics

Suspender

- In one of three states
  - **Inactive** - not being used at the moment
  - **Active**[*caller*] - control is inside the Suspender, with *caller* being the function that called into the Suspender and is expecting an externref to be returned
  - **Suspended** - currently waiting for some promise to resolve

susp.returnPromiseOnSuspend(func)(args)

1. Traps if *susp*'s state is not **Inactive**
2. Changes *susp*'s state to **Active**[*caller*] (where *caller* is the current caller)
3. Calls *func*(*args*) (coercing *args* as necessary)
4. Asserts that *susp*'s state is **Active**[*caller*'] for some *caller*' (should be guaranteed)
5. Changes *susp*'s state to **Inactive**
6. Returns the value returned by *func* to *caller*' (coercing as necessary)

susp.suspendOnReturnedPromis(func)(args)

1. Traps if *susp*'s state is not **Active**[*caller*] for some *caller*
2. Changes *susp*'s state to **Suspended**
3. Calls *func*(*args*) (coercing *args* as necessary)
4. If the value returned is not a Promise, then changes *susp*'s state to **Active**[*caller*] and returns (coerced) value
5. Lets *frames* be the stack frames since *caller*
6. Traps if there are any non-suspendable (e.g. JS) frames in *frames*
7. Returns the result of calling *then(onFulfilled, onRejected)* on the returned Promise with functions *onFulfilled* and *onRejected* that do the following:
   - A. Asserts that *susp*'s state is **Suspended** (should be guaranteed)
   - B. Changes *susp*'s state to **Active**[*caller*'], where *caller*' is the caller of *onFulfilled*/*onRejected*
   - C. In the case of *onFulfilled*, returns the (coerced) given value to *frames*
   - D. In the case of *onRejected*, throws the (coerced) given value up to *frames* as an exception