WasmGC: Minimal JavaScript Interop

a.k.a. the "no-frills" approach for the MVP
– DESIGN DRAFT –

Authors: jkummerow@chromium.org, tlively@google.com, asumu@igalia.com PLEASE ADD YOURSELF HERE (don't be shy!)
Date: March-October 2022

Overview

The WebAssembly Garbage Collection subgroup has decided to split customizable JavaScript interaction (which would e.g. allow accessing struct fields by a module-specified name, or installing JavaScript-visible methods and/or prototypes on objects) to a separate proposal.
For the first version ("MVP") of WasmGC, we will only have a minimal approach to JavaScript interaction: capable enough to express required functionality, but with no importance being placed on ergonomics or prettiness. Of course, forward-compatibility to the aforementioned future extension is important.

Purpose of this document

This document is meant to be a conveniently-editable scratchpad to collect ideas, requirements, wishes, concerns, and to distill them into a first version of a concrete specification of the details of this minimal approach. Once sufficient solidity has been achieved, it will form the basis for a pull request to the WasmGC repository – where of course the details can further evolve as part of the regular designing and decision-making process of the WasmGC proposal.

Detailed design

General principle: keep it minimal!

toWebAssemblyValue

`toWebAssemblyValue` is the procedure in the JS Spec that converts JS values to WebAssembly values when they are passed to exports, returned from imports, assigned to globals, or inserted into tables, etc. This coercion procedure is extended as follows:

```
If `type` is (ref none), (ref nofunc), or (ref noextern)
  Throw a TypeError
```

```
If `type` is nullref, nullfuncref, or nullexternref
  If `v` is null,
    Return ref.null none, ref.null nofunc, or ref.null noextern, respectively
  Throw a TypeError

If `type` is (ref extern)
  If `v` is null,
    Throw a TypeError
  Return toWebAssemblyValue(`v`, externref)

If `type is (ref func)
  If `v` is null,
    Throw a TypeError
  Return toWebAssemblyValue(`v`, funcref)

If `type` is (ref i31)
  If `v` is null,
    Throw a TypeError
  Return toWebAssemblyValue(`v`, i31ref)

If `type` is i31ref
  If `v` is null,
    Return ref.null none
  Return (i31.new (toWebAssemblyValue(`v`, i32))

If `type` <: (ref null? any):
  If `type` is not nullable and `v` is null,
    Throw a TypeError
  Return the result of (ref.cast `type` (extern.internalize (toWebAssemblyValue(`v`,  (ref null?
extern)))))

If `type` <: (ref null? func):
  If `type` is not nullable and `v` is null,
    Throw a TypeError
  Return the result of (ref.cast `type` (toWebAssemblyValue(`v`, (ref null? func)))
```

Wasm objects in JS appear opaque

Wasm structs and arrays will appear on the JavaScript side as frozen empty objects, i.e. having no properties, and it's not possible to add any. Attempting to write a property throws a TypeError.

Formally: their [[Set]], [[DefineOwnProperty]], [[Delete]] internal slots throw exceptions (regardless of strict mode). [[HasProperty]] and [[IsExtensible]] return false. [[Get]] and [[GetOwnProperty]] return undefined. [[OwnPropertyKeys]] returns an empty list.

Note: recent update: we decided that [[Get]] should return `undefined` instead of throwing an exception, because we were worried about breaking existing code that assumes that checking for presence of a property is safe. A particular example that's even built into the JS spec is Promise.resolve(x), which looks up `x.then`, and would make it impossible to resolve promises with Wasm values if their [[Get]] trap always threw an exception. It stands to reason that there might be nontrivial amounts of existing userland code following similar patterns.

The prototype of Wasm objects is null and cannot be changed

[[GetPrototypeOf]] returns `null`. [[SetPrototypeOf]] throws. The former is mandated by making [[Get]] property lookups work; the latter reflects the fact that aside from this constraint we would ideally have liked to disallow prototype access entirely, for alignment with JS Shared Structs and reducing forwards compatibility risk. ~~Attempting to access the [[Prototype]] slot of a Wasm object throws.~~
Note: We could have chosen to install an immutable default prototype instead, which would have given us a place to put helpers. However, this creates forward compatibility risk with potential future extensions that want to enable customized prototypes.
~~Note: We could have chosen to have an immutable `null` prototype. The primary reason for making the prototype inaccessible instead is alignment with Shared Structs.~~
Note: This document earlier suggested that all prototype accesses should throw, for alignment with JS Shared Structs (specifically this). However, when we found that [[Get]] should work, we also had to make [[GetPrototypeOf]] return `null` without throwing.

Identity

Structs and arrays have object identity, i.e. `obj === obj` returns true, and handing the same Wasm-side object to JavaScript multiple times preserves this property (i.e. the JavaScript-side results of all these handovers are `===`-equal to each other).

Note: For Wasm functions this is already the case, and won't change. Arbitrary JavaScript functions, being JS objects, can be round-tripped through Wasm as `externref` just like any other JS object; they'll be opaque references on the Wasm side in that case.

typeof, @@toStringTag, and other tidbits

As a rule of thumb, a Wasm struct/array will behave as if created by `obj = Object.freeze(Object.create(null))`. In particular:

`typeof` will return `"object"` for Wasm structs/arrays. This is meant to be a safe default. There are some estimates that returning a distinguishable value might be handy, but giving a special kind of object its own typeof would be a first in JS and as such would need a stronger argument than "might be handy".

As a consequence of the prototype being null (see above), Wasm structs/arrays will have neither `valueOf` nor `toString`, so e.g. `wasm_object.toString()` will throw. Stringification with `Object.prototype.toString.call(wasm_object)` will return `"[object Object]"`.
An attempt to convert implicitly, like `"" + wasm_object`, throws an exception.

Note: These details are generally negotiable, if sufficiently strong arguments are presented.
Note: Existing JavaScript features can already create objects that throw in these situations, so robust JS code (e.g. for error message formatting) already must be able to handle that.

Wasm objects can be used as Map/Set keys

Wasm structs and arrays can be used as keys in JavaScript Maps, Sets, WeakMaps, FinalizationRegistries.

Note: These collections use hashing, and we expect that this support may have a not-insignificant memory cost (around 10%) in initial implementations, which may have to store a hash key with each object. There are a couple of approaches how engines can avoid this (e.g. certain side tables, or a heap/GC design that keeps immovable references to objects). Should it turn out that many engines are unable to implement this efficiently in the long run, we might discuss alternatives, such as a future Wasm object type that is explicitly un-hash-able (which may be difficult to spec), or a way to recover the cost by exposing the internal hash field to Wasm code to avoid the need of having an additional module-defined hash field in them (which won't benefit all modules and might be nontrivial for some engines).

Wasm Tables

While this isn't officially specified yet, we expect that Wasm tables will be able to hold arbitrary ref-types (in addition to the `funcref` and `externref` values they can hold today).
When reading/writing elements of such Wasm tables from JS, `extern.externalize`/`extern.internalize` is performed implicitly. (This is in contrast to what function calls allow).

Rationale: Table manipulations are likely less frequent and hence less performance relevant than function calls. Also, contrary to functions there's no obvious alternative place where explicit casts could be performed.

Wasm Globals

Same as for Tables: Globals can hold arbitrary ref-types, and externalize/internalize is performed implicitly.

i31ref values

An i31ref value will appear as a primitive number on the JavaScript side, even when it is passed across the boundary as an `externref`. In other words, passing the result of `(i31.new (i32.const 42))` to JavaScript leads to the same result as e.g. `let x = 42`.
When going from JS to Wasm and sending a JS value through `extern.internalize`, any JS Number that can be represented as i31ref will be converted to i31ref.

Note: The reason for requiring conversions to i31ref is that the alternatives are worse: we'd either have to require the opposite, i.e. disallow all JS Numbers from appearing to be i31refs, which would require boxing those that used the same tagged representation; or we would have to allow random behavior there ("a JS Number in i31ref may or may not appear to be an i31ref"), which Wasm generally avoids.
Note: A notable case is -0, which must remain a JS Number (opaque to Wasm) to maintain the property that `extern.internalize`+`extern.externalize` round trips don't change a value. Similarly, JS BigInts are not converted even when in range, because it would be impossible to faithfully undo that (because an i31ref(42) wouldn't remember whether it came from a Number or a BigInt with that value).

Open Questions

Conversion functions for Wasm array <-> ArrayBuffer?

We may want to offer built-in facilities for quick (copying) conversion between Wasm arrays (with primitive/numeric element types) and ArrayBuffers and/or TypedArrays and/or regular JS Arrays. Please speak up if you have an urgent need for this.
Such functionality would likely live on the `WebAssembly` object (e.g. `let buffer = WebAssembly.Array.toArrayBuffer(my_array)`).

Note: Specifying Wasm arrays to be actual ArrayBufferViews (including the possibility to create other views of the same underlying buffer and mutually observing changes) isn't going to happen.
Note: If we had prototypes, we could install helpers there, e.g. `my_array.toArrayBuffer()`, but since we'll go with `null` prototypes for now, the helpers have to live elsewhere.
Note: Since such helpers aren't entangled with anything else, they could easily be shipped separately, e.g. as a soon-after-MVP follow-up proposal.

FAQ

How to pass JS values to Wasm?

JS values can be round-tripped as `externref` through Wasm and will come back as the same object. Wasm cannot inspect the details of such objects (with i31ref being an exception to this rule).

Note: The extern.internalize instruction is infallible, i.e. it will convert any value's type to anyref, but for arbitrary JS objects these anyrefs will fail any subsequent type checks, so they will remain opaque references.

How to access struct fields and array elements from JS?

JavaScript cannot access object contents directly, but the Wasm module can export functions to be used as getters/setters. This doesn't require us to specify any new features.

Note: Medium-term, engines would be expected to optimize this pattern, which likely means inlining such tiny Wasm accessor functions, so that this solution shouldn't be slower than alternatives offering direct syntax.

What is this whole `externref` + `extern.internalize` business anyway?

As the name suggests, `externref` is for reference values that are external (and opaque) to Wasm. This is already the case (as of the reference types proposal, which has been finalized), and isn't changing. For example, you can use it to let Wasm code hold on to JS objects or DOM nodes or whatever; Wasm code won't be able to look at the details of these objects in any way, but it can pass them back to the JS world when needed.
The new reference types that the GC proposal adds are not subtypes of `externref`, they are in their own hierarchy, with `anyref` being their top type, and all conversions between `externref` and `anyref` are explicit. (Note: the name "anyref" might change, it is historical and has become somewhat misleading as the proposal evolved.) The reason for this split is that engines might need to represent the same object differently in the JS and Wasm worlds, and changing between these representations then incurs a cost, so the design makes this explicit and lets modules decide whether they want to pay this price or not.
Concretely:

- A Wasm module that only needs to store opaque references to external values can simply use `externref`, and never bother with `extern.internalize`.
- The purpose of `extern.internalize` is to support the mirrored pattern: the Wasm module might produce values, hand them out to JavaScript, which considers them opaque references (per the rest of this document), and at some point hands them back to the Wasm module, which then likely needs to cast them back to their original type in order to do something useful with them. That's what `extern.internalize` is useful for.

For the current WasmGC feature set, in V8's implementation `extern.internalize` is fairly cheap but not entirely free, `extern.externalize` is a no-op under the hood. It is possible that future additions of functionality (or different/future design choices in engines) might force these instructions to do slightly more work.

How can I tell whether a JavaScript variable is holding a Wasm object or a JS object?

If you really need to distinguish JS and Wasm objects, you can export a function from your Wasm module that performs a type check, based on `extern.internalize` + `ref.test <struct>` + `ref.test <array>`.

Note: If this turns out to be a common requirement, we could add a more convenient way to do this.

Out Of Scope

The following features are deliberately and definitely not part of the minimal approach:

- customizable prototypes
    - unclear where/how these would be specified, unclear need
- custom names for struct fields
    - unclear where/how these would be specified
    - if specified by JS: would probably add unacceptable slowdowns to page load critical path
    - if specified by Wasm module: unclear how to resolve conflicts when multiple modules define conflicting names for isorecursively-canonicalized types
- direct access to array elements and/or length
    - likely no performance benefit over "exported accessor" approach
    - would create an expectation that Wasm arrays can be used everywhere where "array-like" JS objects are accepted, which would be *a lot* of implementation effort for engines for unclear benefit
- methods
    - unclear where/how these would be specified, unclear need
- ability to define Wasm types in JavaScript
    - unclear how to do this without massive instantiation slowdown; unclear how to integrate with isorecursive hybrid types
- direct construction of Wasm objects from JavaScript. If JavaScript code wants to create Wasm structs/arrays, the Wasm module must export functions that do that. Note: this will hence not use `new` syntax in JavaScript, instead e.g. `let my_foo = module.exports.make_foo(arg1, arg2)`.

—

The attic of obsolete paragraphs:

Passing a number from JS into Wasm where an i31ref is expected should be equivalent to calling `(i31.new X)` where X is the result of passing that number where an i31 would be expected. (TODO: how to state this more succinctly/formally?)