JS Primitive Builtins

Wasm CG, October 28, 2025

Background: JS string builtins (Phase 4)

- Reuse the string implementation of the JS host
- Interoperate with JS APIs that accept and return strings

```
(import "wasm:js-string" "test" (func $test (param externref) (result i32)))
(import "wasm:js-string" "fromCodePoint" (func $fromCodePoint (param i32) (result (ref extern))))
(import "wasm:js-string" "codePointAt" (func $codePointAt (param externref) (param i32) (result i32)))

(func $foo (param $cp i32) (result i32)
   (local $s (ref extern))
   (local.set $s (call $fromCodePoint (local.get $cp)))
   (drop (call $test (local.get $s))) ;; 1 (true)
   (call $codePointAt (local.get $s) 0) ;; get back $cp
)
```

Background: JS string builtins (2)

- No changes to core Wasm
- Defined in the JS embedder API
- Polyfillable: everything can be written by hand
- More efficient:
  - skip the Wasm-to-JS call overhead
  - skip (some) dynamic type tests

(also an API for string constants, but not relevant today)

Recap: Phase 1 Motivation (1)

```
def foo(x: Int, y: Int): Int = {
  val a = js.Array(x, y)
  val ta = new Int32Array(a)
  ta(0)
}
```

```
// JavaScript helpers
"1": ((x, y) => [x, y]),
"2": ((x) => new Int32Array(x)),
"3": ((x) => x[0]),
```

```
(import "helpers" "1" (func $helper1 (param i32) (param i32) (result (ref any))))
(import "helpers" "2" (func $helper2 (func (param anyref) (result anyref))))
(import "helpers" "3" (func $helper3 (func (param anyref) (result i32)))
```

```
(func $foo (param $x i32) (param $y i32) (result i32)
  (local $a (ref any)) (local $ta anyref)
  (local.set $a (call $helper1 (local.get $x) (local.get $y)))
  (local.set $ta (call $helper2 (local.get $a)))
  (call $helper3 (local.get $ta))
)
```

`ToJSValue` and `ToWebAssemblyValue` do the right thing.

Recap: Phase 1 Motivation (2)

```
def foo(x: Int, y: Int): Int = {
  val a = makeArray2(x, y)
  val ta = new Int32Array(a)
  getTAFirstElem(ta)
}
def makeArray2[T](x: T, y: T): js.Array[T] =
  js.Array(x, y)
```

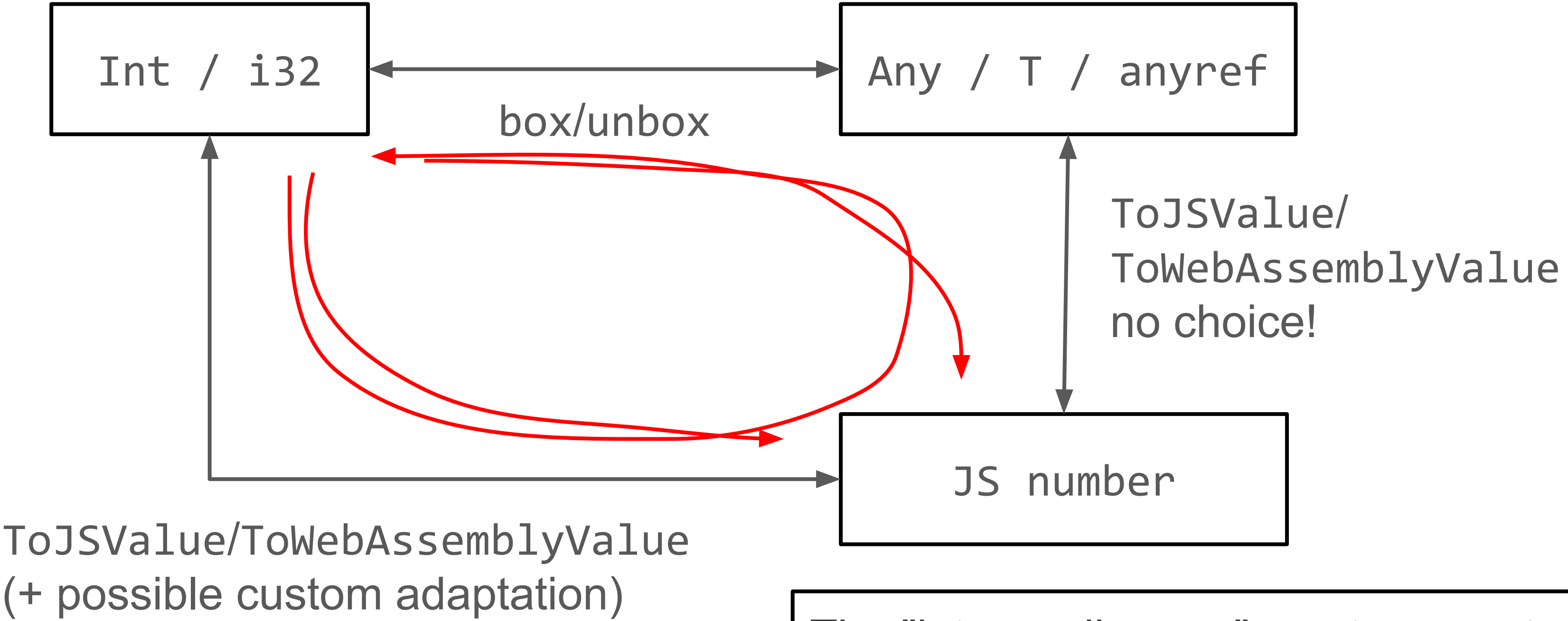
```
// JavaScript helpers
"1": ((x, y) => [x, y]),
"2": ((x) => new Int32Array(x)),
"3": ((x) => x[0]),
```

```
(import "helpers" "1" (func $helper1 (param anyref) (param anyref) (result (ref any))))
(import "helpers" "2" (func $helper2 (func (func (param anyref) (result anyref))))
(import "helpers" "3" (func $helper3 (func (param anyref) (result anyref)))
```

```
(func $foo (param $x i32) (param $y i32) (result i32)
  ...
  (local.set $a (call $makeArray2
    (call $box (local.get $x)) (call $box (local.get $y))))
```

`ToJSValue` and `ToWebAssemblyValue` break the box!

Universal representation and JS interop



Int / i32

Any / T / anyref

box/unbox

ToJSValue/
ToWebAssemblyValue
no choice!

JS number

ToJSValue/ToWebAssemblyValue
(+ possible custom adaptation)

The "interop diagram" must commute.

box and unbox that can commute

```
boxInt: (x) => x,
unboxInt: (x) => x,
```

```
(import "helpers" "boxInt" (func $boxInt (param i32) (result anyref)))
(import "helpers" "unboxInt" (func $unboxInt (param anyref) (result i32)))
```

Semantically correct, but the Wasm-to-JS call is very expensive
(actually shows up high in profiles)

-> This proposal: provide them as JS builtins

box/unbox/test for i32

```
"wasm:js-number" "fromI32"
func fromI32(
  x: i32
) -> (ref extern) {
  return x;
}
```

```
"wasm:js-number" "toI32"
func toI32(
  x: externref
) -> i32 {
  if (typeof x !== "number")
    trap();
  if (!Object.is(x | 0, x))
    trap();
  return x;
}
```

```
"wasm:js-number" "testI32"
func testI32(
  x: externref
) -> i32 {
  if (typeof x !== "number")
    return 0;
  if (!Object.is(x | 0, x))
    return 0;
  return 1;
}
```

Specified in the style of JS string builtins overview for now
(the actual spec text looks different)

Essentials

- `"wasm:js-number"`: `fromX`, `toX` and `testX` for `X` in `{I32`, `U32`, `F64}`
- `"wasm:js-boolean"`: `toI32`, `test`
- `"wasm:js-undefined"`: `test`
- `"wasm:js-symbol"`: `test`
- `"wasm:js-bigint"`: `test`

Still the main focus of the proposal; survived Phase 1 discussions.

Nice-to-have's – initial considered set

- `"wasm:js-number"`
  - JS operators `x % y` (`fmod`) and `x | 0` (`wrapToI32`) – possibly even Wasm core?
- `"wasm:js-bigint"`
  - convert to/from `i64`, `u64`, `f64`, byte arrays, strings
  - full set of operations (available as JS operators)
- `"wasm:js-symbol"`
  - creation as if by `Symbol(description)` or `Symbol.for(key)`
  - extraction of the `description` and the `key`
  - identity test
- `"wasm:js-string"`
  - conversion from all the primitive numeric types
  - parsing into `f64` (= `parseFloat` in JS)
  - `toLowerCase`/`toUpperCase`
  - why those? because they require big tables

Nice-to-have's – almost all thrown away during Phase 1

- `"wasm:js-number"`
  - JS operators `x % y` (`fmod`) and `x | 0` (`wrapToI32`) – possibly even Wasm core?
- `"wasm:js-bigint"`
  - convert to/from `i64`, `u64`, `f64`, byte arrays, strings
  - full set of operations (available as JS operators)
- `"wasm:js-symbol"`
  - creation as if by `Symbol(description)` or `Symbol.for(key)`
  - extraction of the `description` and the key
  - identity test – still an open question
- `"wasm:js-string"`
  - conversion from all the primitive numeric types
  - parsing into `f64` (= `parseFloat` in JS)
  - `toLowerCase`/`toUpperCase`
  - why those? because they require big tables

Controversial – initial considered set

- Universal equality (`Object.is` aka `SameValue`)
- Universal conversion to string (`""+x` aka `ToString`)
    - Can execute arbitrary JS code under the hood!
- Math methods (like `Math.sin`)
    - Not necessary because they're software-defined anyway?

Controversial – thrown away during Phase 1

- Universal equality (`Object.is` aka `SameValue`)
- Universal conversion to string (`""+x` aka `ToString`)
  - Can execute arbitrary JS code under the hood!
- Math methods (like `Math.sin`)
  - Not necessary because they're software-defined anyway?

Alternatives and open questions

`externref` or `anyref`

- JS string builtins set a precedent with `externref`
- Producers will want `anyref` for the main use cases
- Which one do we choose?
- Experimentation could settle this, if no performance cost to using `externref` and converting back and forth

A single "js-value" "fromWasm"

The "box" functions all have the same JS spec: an identity.

The real job is done by `ToJSValue`.

-> Could we have a single builtin that performs `ToJSValue`?

```
"wasm:js-value" "fromWasm"
func fromWasm<T>(
  x: T
) -> (ref null? extern) {
  return x;
}
```

- Needs polymorphic builtins (currently all builtins have a single signature)
- Does not work for unsigned integers
- IMO having 3-4 specific builtins has a lower spec and implementation footprint

A single "js-number" "test"/"cast" (toF64)

Semantically, testI32 and testU32 can be expressed with testF64 + instructions on the Wasm side.

Likewise for toI32 and toU32 with toF64.

Suggested alternative: only expose "test"/"cast"; rely on engines recognizing the shape of the tests afterwards to optimize them away.

```
local.get $x ;; the externref we want to test
call $numberTest
if
  local.get $x ;; same as passed to $numberTest above
  call $numberCast
  local.tee $y ;; a fresh f64
  i32.trunc_sat_f64_s
  f64.convert_i32_s
  i64.reinterpret_f64
  local.get $y
  i64.reinterpret_f64
  i64.eq ;; use an i64.eq test to reject -0.0
else
  i32.const 0
end
```

What qualifies as "directly importable"?

```
const imports = {
  "parseFloat": parseFloat,
  "is": Object.is,
  "toLowerCase": (s) => s.toLowerCase(),
  "toLowerCaseMagic": Function.prototype.call.bind(String.prototype.toLowerCase),
};
```

```
(s) => Function.prototype.call.call(String.prototype.toString, s)
```

```
(s) => String.prototype.toString.call(s)
```

Discussion and Phase 2 Poll

Entry requirements:

- Precise and complete overview document is available in a forked repo around which a reasonably high level of consensus exists.