developer.mozilla.org

Understanding WebAssembly text format - WebAssembly | MDN

29-36 minutes

To enable WebAssembly to be read and edited by humans, there is a textual representation of the wasm binary format. This is an intermediate form designed to be exposed in text editors, browser developer tools, etc. This article explains how that text format works, in terms of the raw syntax, and how it is related to the underlying bytecode it represents — and the wrapper objects representing wasm in JavaScript.

Note: This is potentially overkill if you are a web developer who just wants to load a wasm module into a page and use it in your code (see Using the WebAssembly JavaScript API), but it is more useful if for example, you want to write wasm modules to optimize the performance of your JavaScript library, or build your own WebAssembly compiler.

S-expressions

In both the binary and textual formats, the fundamental unit of code in WebAssembly is a module. In the text format, a module is represented as one big S-expression. S-expressions are a very old and very simple textual format for representing trees, and thus we can think of a module as a tree of nodes that describe the module's structure and its code. Unlike the Abstract Syntax Tree of a programming language, though, WebAssembly's tree is pretty flat, mostly consisting of lists of instructions.

First, let's see what an S-expression looks like. Each node in the tree goes inside a pair of parentheses — `( ... )`. The first label inside the parenthesis tells you what type of node it is, and after that there is a space-separated list of either attributes or child nodes. So that means the WebAssembly S-expression:

```
(module (memory 1) (func))
```

represents a tree with the root node "module" and two child nodes, a "memory" node with the attribute "1" and a "func" node. We'll see shortly what these nodes actually mean.

The simplest module

Let's start with the simplest, shortest possible wasm module.

This module is totally empty, but is still a valid module.

If we convert our module to binary now (see Converting WebAssembly text format to wasm), we'll see just the 8 byte module header described in the binary format:

```
0000000: 0061 736d              ; WASM_BINARY_MAGIC
```

```
0000004: 0100 0000                ; WASM_BINARY_VERSION
```

Adding functionality to your module

Ok, that's not very interesting, let's add some executable code to this module.

All code in a webassembly module is grouped into functions, which have the following pseudo-code structure:

```
( func <signature> <locals> <body> )
```

- The **signature** declares what the function takes (parameters) and returns (return values).
- The **locals** are like vars in JavaScript, but with explicit types declared.
- The **body** is just a linear list of low-level instructions.

So this is similar to functions in other languages, even if it looks different because it is an S-expression.

Signatures and parameters

The signature is a sequence of parameter type declarations followed by a list of return type declarations. It is worth noting here that:

- The absence of a `(result)` means the function doesn't return anything.
- In the current iteration, there can be at most 1 return type, but later this will be relaxed to any number.

Each parameter has a type explicitly declared; wasm currently has four available number types (plus reference types; see the Reference types) section below):

- `i32`: 32-bit integer
- `i64`: 64-bit integer
- `f32`: 32-bit float
- `f64`: 64-bit float

A single parameter is written `(param i32)` and the return type is written `(result i32)`, hence a binary function that takes two 32-bit integers and returns a 64-bit float would be written like this:

```
(func (param i32) (param i32) (result f64) ... )
```

After the signature, locals are listed with their type, for example `(local i32)`. Parameters are basically just locals that are initialized with the value of the corresponding argument passed by the caller.

Getting and setting locals and parameters

Locals/parameters can be read and written by the body of the function with the `local.get` and `local.set` instructions.

The `local.get`/`local.set` commands refer to the item to be got/set by its numeric index: parameters are referred to first, in order of their declaration, followed by locals in order of their declaration. So given the following function:

```
(func (param i32) (param f32) (local f64)
  local.get 0
  local.get 1
  local.get 2)
```

The instruction `local.get 0` would get the i32 parameter, `local.get 1` would get the f32 parameter, and `local.get 2` would get the f64 local.

There is another issue here — using numeric indices to refer to items can be confusing and annoying, so the text format allows you to name parameters, locals, and most other items by including a name prefixed by a dollar symbol ($) just before the type declaration.

Thus you could rewrite our previous signature like so:

```
(func (param $p1 i32) (param $p2 f32) (local $loc f64) …)
```

And then could write `local.get $p1` instead of `local.get 0`, etc. (Note that when this text gets converted to binary, though, the binary will contain only the integer.)

Stack machines

Before we can write a function body, we have to talk about one more thing: **stack machines**. Although the browser compiles it to something more efficient, wasm execution is defined in terms of a stack machine where the basic idea is that every type of instruction pushes and/or pops a certain number of `i32`/`i64`/`f32`/`f64` values to/from a stack.

For example, `local.get` is defined to push the value of the local it read onto the stack, and `i32.add` pops two `i32` values (it implicitly grabs the previous two values pushed onto the stack), computes their sum (modulo 2^32) and pushes the resulting i32 value.

When a function is called, it starts with an empty stack which is gradually filled up and emptied as the body's instructions are executed. So for example, after executing the following function:

```
(func (param $p i32)
  (result i32)
  local.get $p
  local.get $p
  i32.add)
```

The stack contains exactly one `i32` value — the result of the expression (`$p + $p`), which is handled by `i32.add`. The return value of a function is just the final value left on the stack.

The WebAssembly validation rules ensure the stack matches exactly: if you declare a `(result f32)`, then the stack must contain exactly one `f32` at the end. If there is no result type, the stack must be empty.

Our first function body

As mentioned before, the function body is a list of instructions that are followed as the function is called.

Putting this together with what we have already learned, we can finally define a module containing our own simple function:

```
(module
  (func (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add))
```

This function gets two parameters, adds them together, and returns the result.

There are a lot more things that can be put inside function bodies, but we will start off simple for now, and you'll see a lot more examples as you go along. For a full list of the available opcodes, consult the webassembly.org Semantics reference.

Calling the function

Our function won't do very much on its own — now we need to call it. How do we do that? Like in an ES2015 module, wasm functions must be explicitly exported by an `export` statement inside the module.

Like locals, functions are identified by an index by default, but for convenience, they can be named. Let's start by doing this — first, we'll add a name preceded by a dollar sign, just after the `func` keyword:

Now we need to add an export declaration — this looks like so:

```
(export "add" (func $add))
```

Here, `add` is the name the function will be identified by in JavaScript whereas `$add` picks out which WebAssembly function inside the Module is being exported.

So our final module (for now) looks like this:

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add)
  (export "add" (func $add))
)
```

If you want to follow along with the example, save the above our module into a file called `add.wat`, then convert it into a binary file called `add.wasm` using wabt (see Converting WebAssembly text format to wasm for details).

Next, we'll load our binary into a typed array called `addCode` (as described in Fetching WebAssembly Bytecode), compile and instantiate it, and execute our `add` function in JavaScript (we can now find `add()` in the `exports` property of the instance):

```
WebAssembly.instantiateStreaming(fetch('add.wasm'))
  .then(obj => {
    console.log(obj.instance.exports.add(1, 2));  // "3"
  });
```

Exploring fundamentals

Now we've covered the real basics, let's move on to look at some more advanced features.

Calling functions from other functions in the same module

The `call` instruction calls a single function, given its index or name. For example, the following module contains two functions — one just returns the value 42, the other returns the result of calling the first plus one:

```
(module
  (func $getAnswer (result i32)
    i32.const 42)
  (func (export "getAnswerPlus1") (result i32)
    call $getAnswer
    i32.const 1
    i32.add))
```

Note: `i32.const` just defines a 32-bit integer and pushes it onto the stack. You could swap out the `i32` for any of the other available types, and change the value of the const to whatever you like (here we've set the value to 42).

In this example you'll notice an `(export "getAnswerPlus1")` section, declared just after the `func` statement in the second function — this is a shorthand way of declaring that we want to export this function, and defining the name we want to export it as.

This is functionally equivalent to including a separate function statement outside the function, elsewhere in the module in the same manner as we did before, e.g.:

```
(export "getAnswerPlus1" (func $functionName))
```

The JavaScript code to call our above module looks like so:

```
WebAssembly.instantiateStreaming(fetch('call.wasm'))
  .then(obj => {
    console.log(obj.instance.exports.getAnswerPlus1());  // "43"
  });
```

Importing functions from JavaScript

We have already seen JavaScript calling WebAssembly functions, but what about WebAssembly calling JavaScript functions? WebAssembly doesn't actually have any built-in knowledge of JavaScript, but it does have a general way to import functions that can accept either JavaScript or wasm functions. Let's look at an example:

```
(module
  (import "console" "log" (func $log (param i32)))
  (func (export "logIt")
    i32.const 13
    call $log))
```

WebAssembly has a two-level namespace so the import statement here is saying that we're asking to import the `log` function from the `console` module. You can also see that the exported `logIt` function calls the imported function using the `call` instruction we introduced above.

Imported functions are just like normal functions: they have a signature that WebAssembly validation checks statically, and they are given an index and can be named and called.

JavaScript functions have no notion of signature, so any JavaScript function can be passed, regardless of the import's declared signature. Once a module declares an import, the caller of `WebAssembly.instantiate()` must pass in an import object that has the corresponding properties.

For the above, we need an object (let's call it `importObject`) such that `importObject.console.log` is a JavaScript function.

This would look like the following:

```
var importObject = {
  console: {
    log: function(arg) {
      console.log(arg);
    }
  }
};

WebAssembly.instantiateStreaming(fetch('logger.wasm'), importObject)
  .then(obj => {
    obj.instance.exports.logIt();
  });
```

Declaring globals in WebAssembly

WebAssembly has the ability to create global variable instances, accessible from both JavaScript and importable/exportable across one or more `WebAssembly.Module` instances. This is very useful, as it allows dynamic linking of multiple modules.

In WebAssembly text format, it looks something like this (see global.wat in our GitHub repo; also see global.html for a live JavaScript example):

```
(module
   (global $g (import "js" "global") (mut i32))
   (func (export "getGlobal") (result i32)
        (global.get $g))
   (func (export "incGlobal")
        (global.set $g
            (i32.add (global.get $g) (i32.const 1))))
)
```

This looks similar to what we've seen before, except that we specify a global value using the keyword `global`, and we also specify the keyword `mut` along with the value's datatype if we want it to be mutable.

To create an equivalent value using JavaScript, you'd use the `WebAssembly.Global()` constructor:

```
const global = new WebAssembly.Global({value: "i32", mutable: true}, 0);
```

WebAssembly Memory

The above example is a pretty terrible logging function: it only prints a single integer! What if we wanted to log a text string? To deal with strings and other more complex data types, WebAssembly provides **memory** (although we also have Reference types in newer implementation of WebAssembly). According to WebAssembly, memory is just a large array of bytes that can grow over time. WebAssembly contains instructions like `i32.load` and `i32.store` for reading and writing from linear memory.

From JavaScript's point of view, it's as though memory is all inside one big (resizable) `ArrayBuffer`. That's literally all that asm.js has to play with (except that it isn't resizable; see the asm.js Programming model).

So a string is just a sequence of bytes somewhere inside this linear memory. Let's assume that we've written a suitable string of bytes to memory; how do we pass that string out to JavaScript?

The key is that JavaScript can create WebAssembly linear memory instances via the `WebAssembly.Memory()` interface, and access an existing memory instance (currently you can only have one per module instance) using the associated instance methods. Memory instances have a `buffer` getter, which returns an `ArrayBuffer` that points at the whole linear memory.

Memory instances can also grow, for example via the `Memory.grow()` method in JavaScript. When growth occurs, since `ArrayBuffers` can't change size, the current `ArrayBuffer` is detached and a new `ArrayBuffer` is created to point to the newer, bigger memory. This means all we need to do to pass a string to JavaScript is to pass out the offset of the string in linear memory along with some way to indicate the length.

While there are many different ways to encode a string's length in the string itself (for example, C strings); for simplicity here we just pass both offset and length as parameters:

```
(import "console" "log" (func $log (param i32) (param i32)))
```

On the JavaScript side, we can use the TextDecoder API to easily decode our bytes into a JavaScript string. (We specify `utf8` here, but many other encodings are supported.)

```
function consoleLogString(offset, length) {
  var bytes = new Uint8Array(memory.buffer, offset, length);
  var string = new TextDecoder('utf8').decode(bytes);
  console.log(string);
}
```

The last missing piece of the puzzle is where `consoleLogString` gets access to the WebAssembly `memory`. WebAssembly gives us a lot of flexibility here: we can either create a `Memory` object in JavaScript and have the WebAssembly module import the memory, or we can have the WebAssembly module create the memory and export it to JavaScript.

For simplicity, let's create it in JavaScript then import it into WebAssembly. Our `import` statement is written as follows:

```
(import "js" "mem" (memory 1))
```

The `1` indicates that the imported memory must have at least 1 page of memory (WebAssembly defines a page to be 64KB.)

So let's see a complete module that prints the string “Hi”. In a normal compiled C program, you’d call a function to allocate some memory for the string, but since we’re just writing our own assembly here and we own the entire linear memory, we can just write the string contents into global memory using a `data` section. Data sections allow a string of bytes to be written at a given offset at instantiation time and are similar to the `.data` sections in native executable formats.

Our final wasm module looks like this:

```
(module
  (import "console" "log" (func $log (param i32 i32)))
  (import "js" "mem" (memory 1))
  (data (i32.const 0) "Hi")
  (func (export "writeHi")
    i32.const 0  ;; pass offset 0 to log
    i32.const 2  ;; pass length 2 to log
    call $log))
```

Note: Above, note the double semi-colon syntax (`;;`) for allowing comments in WebAssembly files.

Now from JavaScript we can create a Memory with 1 page and pass it in. This results in "Hi" being printed to the console:

```
var memory = new WebAssembly.Memory({initial:1});

var importObject = { console: { log: consoleLogString }, js: { mem: memory }
};

WebAssembly.instantiateStreaming(fetch('logger2.wasm'), importObject)
  .then(obj => {
    obj.instance.exports.writeHi();
  });
```

WebAssembly tables

To finish this tour of the WebAssembly text format, let’s look at the most intricate, and often confusing, part of WebAssembly: **tables**. Tables are basically resizable arrays of references that can be accessed by index from WebAssembly code.

To see why tables are needed, we need to first observe that the `call` instruction we saw earlier (see Calling functions from other functions in the same module) takes a static function index and thus can only ever call one function — but what if the callee is a runtime value?

- In JavaScript we see this all the time: functions are first-class values.
- In C/C++, we see this with function pointers.
- In C++, we see this with virtual functions.

WebAssembly needed a type of call instruction to achieve this, so we gave it `call_indirect`, which takes a dynamic function operand. The problem is that the only types we have to give operands in WebAssembly are (currently) `i32`/`i64`/`f32`/`f64`.

WebAssembly could add an `anyfunc` type ("any" because the type could hold functions of any signature), but unfortunately this `anyfunc` type couldn't be stored in linear memory for security reasons. Linear memory exposes the raw contents of stored values as bytes and this would allow wasm content to arbitrarily observe and corrupt raw function addresses, which is something that cannot be allowed on the web.

The solution was to store function references in a table and pass around table indices instead, which are just i32 values. `call_indirect`'s operand can therefore be an i32 index value.

Defining a table in wasm

So how do we place wasm functions in our table? Just like `data` sections can be used to initialize regions of linear memory with bytes, `elem` sections can be used to initialize regions of tables with functions:

```
(module
  (table 2 funcref)
  (elem (i32.const 0) $f1 $f2)
  (func $f1 (result i32)
    i32.const 42)
  (func $f2 (result i32)
    i32.const 13)
  ...
)
```

- In `(table 2 funcref)`, the 2 is the initial size of the table (meaning it will store two references) and `funcref` declares that the element type of these references are function reference.
- The functions (`func`) sections are just like any other declared wasm functions. These are the functions we are going to refer to in our table (for example's sake, each one just returns a constant value). Note that the order the sections are declared in doesn't matter here — you can declare your functions anywhere and still refer to them in your `elem` section.

- The `elem` section can list any subset of the functions in a module, in any order, allowing duplicates. This is a list of the functions that are to be referenced by the table, in the order they are to be referenced.
- The `(i32.const 0)` value inside the `elem` section is an offset — this needs to be declared at the start of the section, and specifies at what index in the table function references start to be populated. Here we've specified 0, and a size of 2 (see above), so we can fill in two references at indexes 0 and 1. If we wanted to start writing our references at offset 1, we'd have to write `(i32.const 1)`, and the table size would have to be 3.

  Note: Uninitialized elements are given a default throw-on-call value.

  In JavaScript, the equivalent calls to create such a table instance would look something like this:

  ```
  function() {
    // table section
    var tbl = new WebAssembly.Table({initial:2, element:"anyfunc"});

    // function sections:
    var f1 = ... /* some imported WebAssembly function */
    var f2 = ... /* some imported WebAssembly function */

    // elem section
    tbl.set(0, f1);
    tbl.set(1, f2);
  };
  ```

  Using the table

  Moving on, now we've defined the table we need to use it somehow. Let's use this section of code to do so:

  ```
  (type $return_i32 (func (result i32))) ;; if this was f32, type checking
  would fail
  (func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32))
  ```

- The `(type $return_i32 (func (result i32)))` block specifies a type, with a reference name. This type is used when performing type checking of the table function reference calls later on. Here we are saying that the references need to be functions that return an `i32` as a result.
- Next, we define a function that will be exported with the name `callByIndex`. This will take one `i32` as a parameter, which is given the argument name `$i`.
- Inside the function, we add one value to the stack — whatever value is passed in as the parameter `$i`.
- Finally, we use `call_indirect` to call a function from the table — it implicitly pops the value of `$i` off

the stack. The net result of this is that the `callByIndex` function invokes the `$i`'th function in the table.

You could also declare the `call_indirect` parameter explicitly during the command call instead of before it, like this:

```
(call_indirect (type $return_i32) (local.get $i))
```

In a higher level, more expressive language like JavaScript, you could imagine doing the same thing with an array (or probably more likely, object) containing functions. The pseudo code would look something like `tbl[i]()`.

So, back to the typechecking. Since WebAssembly is typechecked, and the `funcref` can be potentially any function signature, we have to supply the presumed signature of the callee at the callsite, hence we include the `$return_i32` type, to tell the program a function returning an `i32` is expected. If the callee doesn't have a matching signature (say an `f32` is returned instead), a `WebAssembly.RuntimeError` is thrown.

So what links the `call_indirect` to the table we are calling? The answer is that there is only one table allowed right now per module instance, and that is what `call_indirect` is implicitly calling. In the future, when multiple tables are allowed, we would also need to specify a table identifier of some kind, along the lines of

```
call_indirect $my_spicy_table (type $i32_to_void)
```

The full module all together looks like this, and can be found in our wasm-table.wat example file:

```
(module
  (table 2 funcref)
  (func $f1 (result i32)
    i32.const 42)
  (func $f2 (result i32)
    i32.const 13)
  (elem (i32.const 0) $f1 $f2)
  (type $return_i32 (func (result i32)))
  (func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32))
)
```

We load it into a webpage using the following JavaScript:

```
WebAssembly.instantiateStreaming(fetch('wasm-table.wasm'))
  .then(obj => {
    console.log(obj.instance.exports.callByIndex(0)); // returns 42
    console.log(obj.instance.exports.callByIndex(1)); // returns 13
    console.log(obj.instance.exports.callByIndex(2)); // returns an error,
because there is no index position 2 in the table
  });
```

Note: Just like Memory, Tables can also be created from JavaScript (see `WebAssembly.Table()`) as well as imported to/from another wasm module.

Mutating tables and dynamic linking

Because JavaScript has full access to function references, the Table object can be mutated from JavaScript using the `grow()`, `get()` and `set()` methods. And WebAssembly code is itself able to manipulate tables using instructions added as part of Reference types, such as `table.get` and `table.set`.

Because tables are mutable, they can be used to implement sophisticated load-time and run-time dynamic linking schemes. When a program is dynamically linked, multiple instances share the same memory and table. This is symmetric to a native application where multiple compiled `.dlls` share a single process's address space.

To see this in action, we'll create a single import object containing a Memory object and a Table object, and pass this same import object to multiple `instantiate()` calls.

Our `.wat` examples look like so:

`shared0.wat`:

```
(module
  (import "js" "memory" (memory 1))
  (import "js" "table" (table 1 funcref))
  (elem (i32.const 0) $shared0func)
  (func $shared0func (result i32)
   i32.const 0
   i32.load)
)
```

`shared1.wat`:

```
(module
  (import "js" "memory" (memory 1))
  (import "js" "table" (table 1 funcref))
  (type $void_to_i32 (func (result i32)))
  (func (export "doIt") (result i32)
   i32.const 0
   i32.const 42
   i32.store  ;; store 42 at address 0
   i32.const 0
   call_indirect (type $void_to_i32))
)
```

These work as follows:

1. The function `shared0func` is defined in `shared0.wat`, and stored in our imported table.

2. This function creates a constant containing the value `0`, and then uses the `i32.load` command to load the value contained in the provided memory index. The index provided is `0` — again, it implicitly pops the previous value off the stack. So `shared0func` loads and returns the value stored at memory index `0`.
3. In `shared1.wat`, we export a function called `doIt` — this function creates two constants containing the values `0` and `42`, then calls `i32.store` to store a provided value at a provided index of the imported memory. Again, it implicitly pops these values off the stack, so the result is that it stores the value `42` in memory index `0`,
4. In the last part of the function, we create a constant with value `0`, then call the function at this index 0 of the table, which is `shared0func`, stored there earlier by the `elem` block in `shared0.wat`.
5. When called, `shared0func` loads the `42` we stored in memory using the `i32.store` command in `shared1.wat`.

   Note: The above expressions again pop values from the stack implicitly, but you could declare these explicitly inside the command calls instead, for example:

   ```
   (i32.store (i32.const 0) (i32.const 42))
   (call_indirect (type $void_to_i32) (i32.const 0))
   ```

   After converting to assembly, we then use `shared0.wasm` and `shared1.wasm` in JavaScript via the following code:

   ```
   var importObj = {
     js: {
       memory : new WebAssembly.Memory({ initial: 1 }),
       table : new WebAssembly.Table({ initial: 1, element: "anyfunc" })
     }
   };

   Promise.all([
     WebAssembly.instantiateStreaming(fetch('shared0.wasm'), importObj),
     WebAssembly.instantiateStreaming(fetch('shared1.wasm'), importObj)
   ]).then(function(results) {
     console.log(results[1].instance.exports.doIt());  // prints 42
   });
   ```

   Each of the modules that is being compiled can import the same memory and table objects and thus share the same linear memory and table "address space".

Bulk memory operations

Bulk memory operations are a newer addition to the language (for example, in Firefox 79) — seven new built-in operations are provided for bulk memory operations such as copying and initializing, to allow WebAssembly to model native functions such as `memcpy` and `memmove` in a more efficient, performant way.

The new operations are:

- `data.drop`: Discard the data in an data segment.
- `elem.drop`: Discard the data in an element segment.
- `memory.copy`: Copy from one region of linear memory to another.
- `memory.fill`: Fill a region of linear memory with a given byte value.
- `memory.init`: Copy a region from a data segment.
- `table.copy`: Copy from one region of a table to another.
- `table.init`: Copy a region from an element segment.

Reference types

The reference types proposal (supported in Firefox 79) provides two main features:

- A new type, `externref`, which can hold *any* JavaScript value, for example strings, DOM references, objects, etc. `externref` is opaque from the point of view of WebAssembly — a wasm module can't access and manipulate these values and instead can only receive them and pass them back out. But this is very useful for allowing wasm modules to call JavaScript functions, DOM APIs, etc., and generally to pave the way for easier interoperability with the host environment. `externref` can be used for value types and table elements.
- A number of new instructions that allow wasm modules to directly manipulate WebAssembly tables, rather than having to do it via the JavaScript API.

Note: The wasm-bindgen documentation contains some useful information on how to take advantage of `externref` from Rust.

Multi-value WebAssembly

Another more recent addition to the language (for example, in Firefox 78) is WebAssembly multi-value, meaning that WebAssembly functions can now return multiple values, and instruction sequences can consume and produce multiple stack values.

At the time of writing (June 2020) this is at an early stage, and the only multi-value instructions available are calls to functions that themselves return multiple values. For example:

```
(module
  (func $get_two_numbers (result i32 i32)
    i32.const 1
    i32.const 2
  )
  (func (export "add_two_numbers") (result i32)
    call $get_two_numbers
    i32.add
  )
```

)

But this will pave the way for more useful instruction types, and other things besides. For a useful write up of progress so far and how this works, see Multi-Value All The Wasm! by Nick Fitzgerald.

WebAssembly threads

WebAssembly Threads (supported in Firefox 79 onwards) allow WebAssembly Memory objects to be shared across multiple WebAssembly instances running in separate Web Workers, in the same fashion as `SharedArrayBuffer`s in JavaScript. This allows very fast communication between Workers, and significant performance gains in web applications.

The threads proposal has two parts, shared memories and atomic memory accesses.

Shared memories

As described above, you can create shared WebAssembly `Memory` objects, which can be transferred between Window and Worker contexts using `postMessage()`, in the same fashion as a `SharedArrayBuffer`.

Over on the JavaScript API side, the `WebAssembly.Memory()` constructor's initialization object now has a `shared` property, which when set to `true` will create a shared memory:

```
let memory = new WebAssembly.Memory({initial:10, maximum:100, shared:true});
```

the memory's `buffer` property will now return a `SharedArrayBuffer`, instead of the usual `ArrayBuffer`:

```
memory.buffer // returns SharedArrayBuffer
```

Over in the text format, you can create a shared memory using the `shared` keyword, like this:

Unlike unshared memories, shared memories must specify a "maximum" size, in both the JavaScript API constructor and wasm text format.

Atomic memory accesses

A number of new wasm instructions have been added that can be used to implement higher level features like mutexes, condition variables, etc. You can find them listed here. These instructions are allowed on non-shared memories as of Firefox 80.

Summary

This finishes our high-level tour of the major components of the WebAssembly text format and how they get reflected in the WebAssembly JS API.

See also

- The main thing that wasn't included is a comprehensive list of all the instructions that can occur in function bodies. See the WebAssembly semantics for a treatment of each instruction.

- See also the grammar of the text format that is implemented by the spec interpreter.