`asm.js`: a High-Performance Subset of JavaScript

Dave Herman, Luke Wagner, and Alon Zakai

January 18, 2013

1 Introduction

This document describes a formal definition of a subset of the JavaScript programming language that can be used as a high-performance compiler target language. This sublanguage or dialect, which we call `asm.js`, effectively describes a safe virtual machine for memory-unsafe languages such as C and C++.

Because `asm.js` is a proper subset of JavaScript, both syntactically and semantically, the language is fully defined by a static *validation* judgment, which yields a predicate that determines whether a given JavaScript program is or is not in the subset. No specification of a dynamic semantics is needed, since the behavior of an `asm.js` program is simply defined by its behavior as a JavaScript program.

1.1 Overview

The unit of compilation/validation of `asm.js` is the `asm.js` *module*, which takes the form of a closed JavaScript function beginning with the *prologue directive* [**?**]:

```
"use asm";
```

The presence of this directive serves two purposes. First, it allows JavaScript engines that wish to provide specialized optimizations for `asm.js` to efficiently recognize that the module should be validated as an `asm.js`, without the need for complex, heuristic or concurrent recognition logic. (Since validation requires a non-trivial traversal of the body of the module, it is likely too expensive to speculatively validate *all* JavaScript code during JIT compilation.) Second, by requiring the programmer or code generator to state the intention explicitly that the code should be recognized as `asm.js`, it allows user agents to report validation errors or performance faults to developer consoles.

An `asm.js` module takes three optional parameters: a global object, containing standard ECMAScript libraries, an *FFI object*, containing custom imported functions and constants from external JavaScript, and a JavaScript

`ArrayBuffer` representing a virtualized memory. The module can provide different views of the buffer by using typed array wrappers imported from the environment:

```
function mod(global, foreign, buffer) {
    "use asm";

    var HEAP_I32 = new global.Int32Array(buffer);
    var HEAP_F64 = new global.Float64Array(buffer);
    // ...
}
```

The body of an `asm.js` module consists of any number of function definitions, followed by an *export clause*:

```
return { foo: f, bar: g };
```

If a module only exports a single function, it can do so directly, without the object literal:

```
return foo;
```

1.2 Types

The `asm.js` language is statically typed: every function, variable, and expression has a statically predictable type, according to a type hierarchy covering a subset of JavaScript values (see Section 3). Variables, parameters, and functions are provided with an explicit type bound through a stylized use of JavaScript coercions. This technique was pioneered by the Emscripten compiler [**?**], and is now used by a number of compilers that target JavaScript [**?**, **?**].

For example, the following is a simple function from integers to integers:

```
function id(x) {
    x = x|0;
    return x|0;
}
```

Even though JavaScript provides only double-precision floating-point numbers (doubles) in its data model, the `asm.js` type system enforces that 32-bit integer values—a strict subset of doubles—never overflow to larger doubles. This allows optimizing compilers to represent these values as unboxed integers in 32-bit registers or memory.

Again following the practice established by Emscripten, it is possible to do integer operations such as arithmetic and conditionals by means of coercions:

```
function add1(x) {
    x = x|0;
    return ((x|0)+1)|0;
}
```

While the JavaScript semantics dictates that the addition may overflow to a larger number than a 32-bit integer, the outer coercion ensures that the entire expression results in a 32-bit integer—the same integer that would be produced by a signed, 32-bit addition in a typical assembly language. The `asm.js` type system thus ensures that integer operations can be efficiently compiled by optimizing JavaScript engines to predictable machine instructions.

1.3 Validation, linking, and execution

The `asm.js` validator is defined as a static type system, which can be performed by an optimizing JavaScript engine at the time the module is parsed by the JavaScript engine. (If compilation time is a concern, it can be delayed to runtime by hiding the source code in a string and passed to `eval` or the `Function` constructor.) During this phase, any static validation errors can be reported to a developer console.

After a `asm.js` module is compiled, its evaluation produces a closure with an empty lexical environment. The module can be *linked* by calling the function with an object representing the imported environment and an optional buffer:

```
function mod(global, foreign, buffer) {
    "use asm";
    // ...
    return { f: foo, g: bar };
}

var foreign = {
    consoleDotLog: console.log,
    // ...
};

var buffer = new ArrayBuffer(0x100000);

// link the module
var m = mod(window, foreign, buffer);
```

This linking phase may need to perform additional, dynamic validation. In particular, dynamic validation can fail if, for example, the `Int32Array` function passed in through the environment does not turn out to construct a proper typed array (thereby defeating typed array-based optimizations).

The resulting module object provides access to the exported `asm.js` functions, which have been fully validated (both statically and dynamically) and optimized.

1.4 Notation conventions

The following notation conventions are used in this document. Optional items in a grammar are presented in [square brackets]. Sequences are presented with

a $\overline{\text{horizontal overbar}}$. The empty sequence is denoted by ϵ. Integers and integer literals are ranged over by the metavariables i, j; these may also serve as sequence indices, in which case they are natural numbers. Natural numbers are otherwise ranged over by the metavariables m, n. Floating-point literals are ranged over by the metavariable r.

1.5 Document outline

The remainder of this document proceeds as follows.

2 Abstract syntax

This section specifies the abstract syntax of `asm.js`. The grammar is presented with concrete syntax for conciseness and readability, but should be read as describing the subset of abstract syntax trees produced by a standard JavaScript parser.

We make the following assumptions about canonicalization of `asm.js` abstract syntax trees:

1. Parentheses are ignored in the AST. This allows parentheses to be left out of any of the formal definitions of this spec.
2. Empty statements (`;`) are ignored in the AST. This allows empty statements to be left out of any of the formal definitions of this spec.
3. The identifiers `arguments` and `eval` do not appear in `asm.js` programs. If either of these identifiers appears anywhere, static validation must fail.

In various places in this document, the meta-variables f, g, x, y, and z are used to range over JavaScript identifiers.

We syntactically distinguish integer literals i, j from floating-point literals r. A floating-point literal is a JavaScript number literal that contains a `.` character.

2.1 Modules

An `asm.js` module has the following syntax:

$$
\begin{array}{rcl}
\mathit{mod} & ::= & \texttt{function}\ [f]\texttt{(}[\mathit{global}[, \mathit{foreign}[, \mathit{buffer}]]]\texttt{) \{} \\
& & \quad \texttt{"use asm";} \\
& & \quad \overline{\texttt{var}\ \overline{x\ \texttt{=}\ \mathit{imp}}\texttt{;}} \\
& & \quad \overline{\mathit{fn}_g} \\
& & \quad \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}} \\
& & \quad \mathit{exp} \\
& & \texttt{\}}
\end{array}
$$

The syntax consists of:

1. an optional module name;

2. up to three optional parameters;

3. a `"use asm";` prologue directive;

4. a sequence of import statements;

5. a sequence of function declarations;

6. a sequence of global variable declarations; and

7. a single export statement.

An import expression is either an FFI function binding, a type-annotated (coerced) FFI value, or a heap view:

$$\begin{array}{rcl} \mathit{imp} & ::= & \mathit{global}\texttt{.}x \\ & | & \mathit{global}\texttt{.Math.}x \\ & | & \texttt{new } \mathit{global}\texttt{.}x\texttt{(}\mathit{buffer}\texttt{)} \\ & | & \mathit{foreign}\texttt{.}x\texttt{|0} \\ & | & \texttt{+}\mathit{foreign}\texttt{.}x \\ & | & \mathit{foreign}\texttt{.}x \end{array}$$

(We assume that the metavariables *global*, *foreign*, and *buffer* stand for fixed variables names used consistently throughout the imports.)

A function declaration has the following syntax:

$$\mathit{fn}_f ::= \texttt{function } f\texttt{(}\overline{x}\texttt{) \{ } \overline{x \texttt{ = } \mathit{ann}_x\texttt{;}}\ \overline{\texttt{var } \overline{y \texttt{ = } v}\texttt{;}}\ \mathit{ss}\ \texttt{\}}$$

The syntax consists of type annotations for the parameters, a sequence of local variable declarations, and a sequence of statements.

Type annotations are either `int` or `double` coercions:

$$\mathit{ann}_x ::= x\texttt{|0} \mid \texttt{+}x$$

An export statement returns either a single function or an object literal containing multiple functions:

$$\begin{array}{rcl} \mathit{exp} & ::= & \texttt{return } f\texttt{;} \\ & | & \texttt{return \{ } \overline{x\texttt{:}f}\ \texttt{\};} \end{array}$$

2.2 Statements

The set of legal statements in `asm.js` includes blocks, expression statements, conditionals, returns, loops, `switch` blocks, `break` and `continue`, and labeled

statements:

$$
\begin{array}{rcl}
s & ::= & \texttt{\{}\ ss\ \texttt{\}} \\
& | & e\texttt{;} \\
& | & \texttt{if (}e\texttt{)}\ s \\
& | & \texttt{if (}e\texttt{)}\ s\ \texttt{else}\ s \\
& | & \texttt{return}\ [re]\texttt{;} \\
& | & \texttt{while (}e\texttt{)}\ s \\
& | & \texttt{do}\ s\ \texttt{while (}e\texttt{);} \\
& | & \texttt{for (}[e]\texttt{;}\ [e]\texttt{;}\ [e]\texttt{)}\ s \\
& | & \texttt{switch (}e\texttt{) \{}\ \overline{c}\ [d]\ \texttt{\}} \\
& | & \texttt{break}\ [lab]\texttt{;} \\
& | & \texttt{continue}\ [lab]\texttt{;} \\
& | & lab\texttt{:}\ s \\
\\
ss & ::= & \overline{s}
\end{array}
$$

Return arguments always have their type explicitly manifest: either a `signed` or `double` coercion or a literal:

$$re ::= e\texttt{|0} \mid \texttt{+}e \mid v$$

The contents of `switch` blocks are restricted: in addition to requiring the (optional) `default` clause to be the last clause, each `case` clause is syntactically restricted to contain only literal values:

$$
\begin{array}{rcl}
c & ::= & \texttt{case}\ v\texttt{:}\ ss \\
d & ::= & \texttt{default:}\ ss \\
cd & ::= & c \mid d
\end{array}
$$

2.3 Expressions

Expressions include literals, lvalues, assignments, function calls, unary expressions, binary expressions, and sequence expressions:

$$
\begin{array}{rcl}
e & ::= & v \\
& | & lval \\
& | & lval\ \texttt{=}\ e \\
& | & f\texttt{(}\overline{e}\texttt{)} \\
& | & unop\ e \\
& | & e\ binop\ e \\
& | & e\texttt{?}\,e\texttt{:}e \\
& | & \texttt{(}\overline{e}\texttt{)} \\
\\
unop & ::= & \texttt{+} \mid \texttt{\~{}} \mid \texttt{!} \\
\\
binop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
& | & \texttt{|} \mid \texttt{\&} \mid \texttt{\^{}} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{>>>} \\
& | & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} \mid \texttt{==}
\end{array}
$$

Literals are either doubles or integers:

$$v ::= r \mid i$$

Lvalues are either variables or typed array dereference expressions. The latter requires a mask to force the byte offset into a valid range and a shift to convert the offset into a proper index for the size of the typed array.

$$\mathit{lval} ::= x \mid x\texttt{[}i\texttt{]} \mid x\texttt{[}e\ \texttt{\&}\ \mathit{mask}\texttt{]} \mid x\texttt{[(}e\ \texttt{\&}\ \mathit{mask}\texttt{) >>}\ i\texttt{]}$$

The *mask* is a non-negative integer $2^k - 1$ where $k \in [3, 31]$. The same *mask* must be used consistently throughout the program.

3 Types

The `asm.js` validator relies on a static type system that classifies and constraints the syntax beyond the grammar.

3.1 Expression types

The set of *expression types* classifies the results of expression evaluation and constrains the allowable values of variables.

$$\begin{array}{rcl} \sigma, \tau & ::= & \texttt{fixnum} \mid \texttt{signed} \mid \texttt{unsigned} \mid \texttt{int} \mid \texttt{intish} \\ & \mid & \texttt{double} \mid \texttt{doublish} \mid \texttt{extern} \mid \texttt{unknown} \mid \texttt{void} \end{array}$$

These types are arranged in a subtyping hierarchy, defined by:

$$\begin{array}{rcl} \texttt{fixnum} & <: & \texttt{signed}, \texttt{unsigned} \\ \texttt{signed}, \texttt{unsigned} & <: & \texttt{int}, \texttt{extern} \\ \texttt{double} & <: & \texttt{extern}, \texttt{doublish} \\ \texttt{unknown} & <: & \texttt{intish}, \texttt{doublish} \\ \texttt{int} & <: & \texttt{intish} \end{array}$$

Figure 1 depicts this subtyping hierarchy visually. Note that some types are presented in white boxes and others in gray boxes. The white boxes represent types that may escape into external JavaScript; the gray types are internal to the `asm.js` module and cannot escape. This allows an optimizing implementation to use unboxed representations that would otherwise be illegal. What follows is an explanation of each type.

The `extern` type

This abstract type represents the root of all types that can escape back into ordinary JavaScript. Any type that is a subtype of `extern` must carry enough information in its internal representation in an optimizing virtual machine to faithfully convert back into a dynamic JavaScript value.

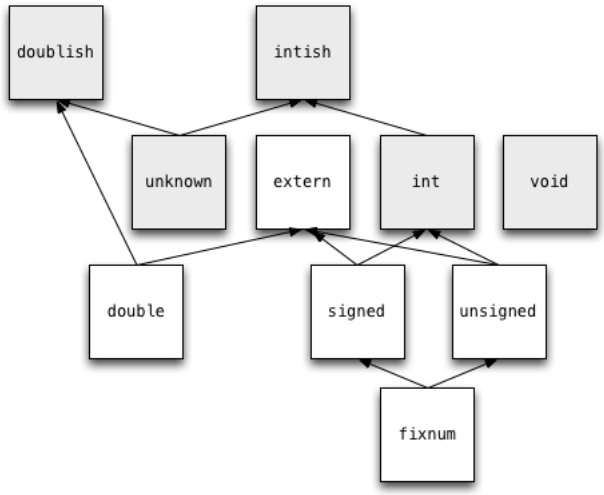

Figure 1: The hierarchy of expression types.

The `double` type

This is the type of double-precision floating-point values. An optimizing engine can represent these as unboxed 64-bit floats. If they escape into external JavaScript they must of course be wrapped back up as JavaScript values according to the JavaScript engine's value representation.

The `signed` and `unsigned` types

These are the types of signed and unsigned 32-bit integers, respectively. For an optimizing engine, their representation can be the same: an unboxed 32-bit integer. If a value escapes into external JavaScript, the sign is used to determine which JavaScript value it represents. For example, the bit pattern `0xffffffff` represents either the JavaScript value `-1` or `4294967295`, depending on the signedness of the type.

The `fixnum` type

This type represents integers in the range $[0, 2^{31})$, which are both valid signed and unsigned integers. Literals in this range are given the type `fixnum` rather than `signed` or `unsigned`, since either way they have the same representation as an unboxed 32-bit integer.

The `int` type

This is the type of 32-bit integers whose sign is not known. Again, optimizing engines can represent them as unboxed 32-bit integers. But because the sign

is not known, they cannot be allowed to escape to external JavaScript, as it is impossible to determine exactly which JavaScript value they represent. While this might not seem like a very useful type, the JavaScript bitwise coercions can be used to force an `int` value back to `signed` or `unsigned` without any loss of data.

The conditional operators also produce the `int` type, even though, according to the JavaScript semantics, they produce boolean values. Since boolean values convert to the integer values `0` and `1`, it is sound to represent the boolean values in an optimizing engine as integers, even while allowing them to be stored in integer values and passed to integer arithmetic. Similarly, integer values can be treated as "boolish" in JavaScript when used in conditional contexts, so it is safe to store boolean values as unboxed integers even when being used in the test expression of an `if` statement and the like.

The `intish` type

The JavaScript arithmetic operations can be performed on 32-bit integer values, but their results may produce non-integer values. For example, addition and subtraction can overflow to large numbers that exceed the 32-bit integer range, and integer division can produce a non-integer value. However, if the result is coerced back to an integer, the resulting arithmetic operation behaves identically to the typical corresponding machine operation (i.e., integer addition, subtraction, or division). The `intish` type represents the result of a JavaScript integer artihmetic operation that must be coerced back to integer with an explicit coercion. Because this type can only be used as an argument to a coercion (or silently ignored in an expression statement), `asm.js` integer arithmetic can always be implemented in an optimizing engine by the machine integer artithmetic operations.

The one arithmetic operation that does not quite fit into this story is multiplication. Multiplying two large integers can results in a large enough double that some lower bits of precision are lost, so that coercing the result back to integer does *not* behave identically to the machine operation. The use of the proposed ES6 `Math.imul` function [**?**] as an FFI function is recommended as the proper means of implementing integer multiplication.

In short:

> The `intish` type represents the result of integer operations that must be dropped or immediately coerced via *ToInt32* or *ToUint32*.

The `doublish` type

Similarly, the `doublish` type represents operations that are expected to produce a `double` but may produce additional junk that must be coerced back to a number via *ToNumber*. In particular, reading from the typed array may produce `undefined`, and calling FFI functions may produce an arbitrary JavaScript value. Thus:

The `doublish` type represents the result of numeric operations that must be dropped or immediately coerced via *ToNumber*.

The `unknown` type

Calling an external JavaScript function through the FFI results in an arbitrary JavaScript value. Because `asm.js` is designed to avoid dealing with general values, the result must be coerced to one of the other types before it can be used. The `unknown` type represents one of these result values before being coerced to an integer or double.

The `void` type

A function that returns `undefined` is considered to have the `void` result type. The `undefined` value is not actually a first-class value in `asm.js`. It can only be ignored via an expression statement. This avoids having to represent it at all as data.

3.2 Global types

Validation tracks the types not only of expressions and local variables but also global variables, FFI imports, and functions. In addition to variables of expression type, globals may also be typed array views on the module's buffer, imported FFI constants and functions, and functions defined in the `asm.js` module.

$$\gamma ::= \tau \mid \texttt{view}_\tau^n \mid ((\overline{\sigma}) \to \tau) \wedge \cdots \wedge ((\overline{\sigma'}) \to \tau') \mid \texttt{Function}$$

The type of a typed array tracks the number of bytes per element and the elements' value type (`intish` or `doublish`). A function type may be overloaded to allow different parameter types, potentially providing different result types for each overloading.

3.3 Operator types

Every operator, unary or binary, has an overloaded function type. This overloading corresponds to the different machine operations used in an optimizing engine to implement the various cases. Whereas JavaScript generally needs to choose the behavior of operators dynamically, `asm.js` makes it possible to resolve the overloaded operators statically based on the types of the operands.

The types of the arithmetic operators are as follows:

$$
\begin{array}{rcl}
\texttt{+} & : & (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\texttt{-} & : & (\texttt{doublish}, \texttt{doublish}) \to \texttt{double} \\
\texttt{*} & : & (\texttt{doublish}, \texttt{doublish}) \to \texttt{double} \\
\texttt{/} & : & (\texttt{doublish}, \texttt{doublish}) \to \texttt{double} \\
 & & \wedge\ (\texttt{signed}, \texttt{signed}) \to \texttt{intish} \\
 & & \wedge\ (\texttt{unsigned}, \texttt{unsigned}) \to \texttt{intish} \\
\texttt{\%} & : & (\texttt{doublish}, \texttt{doublish}) \to \texttt{double} \\
 & & \wedge\ (\texttt{signed}, \texttt{signed}) \to \texttt{int} \\
 & & \wedge\ (\texttt{unsigned}, \texttt{unsigned}) \to \texttt{int}
\end{array}
$$

The type of addition and subtraction only deals with floating-point arithmetic; for integer addition and subtraction see Section 4.11. Similarly, the `*` operator provides only floating-point multiplication; integer multiplication can be implemented via the `imul` function. The `/` operator does provide both floating-point and integer division; for the latter it produces the type `intish`, which requires a coercion via the bitwise operators to produce the proper integer result. The `%` operator works as either a floating-point or integer operator and produces the correct result without any need for a coercion.

The bitwise operators are one of only two places in the language that can consume an `intish` expression. Every one of these operators can be composed with the arithmetic operators to produce the correct behavior of integer arithmetic.

$$
\begin{array}{rcl}
\texttt{|}, \texttt{\&}, \texttt{\^{}}, \texttt{<<}, \texttt{>>} & : & (\texttt{intish}, \texttt{intish}) \to \texttt{signed} \\
\texttt{>>>} & : & (\texttt{intish}, \texttt{intish}) \to \texttt{unsigned}
\end{array}
$$

The conditional operators rely on the sign of their input and produce a boolean result:

$$
\begin{array}{rcl}
\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=} & : & (\texttt{signed}, \texttt{signed}) \to \texttt{int} \\
 & & \wedge\ (\texttt{unsigned}, \texttt{unsigned}) \to \texttt{int} \\
 & & \wedge\ (\texttt{double}, \texttt{double}) \to \texttt{int}
\end{array}
$$

Finally, the unary operators have function types as well; unary operations can also serve as coercions: the `+` operator converts integers and the results of typed array reads and FFI calls to `double`, and the `~` operator can be used to coerce `intish` expressions, similar to the two-argument bitwise operators.

$$
\begin{array}{rcl}
\texttt{+} & : & (\texttt{signed}) \to \texttt{double} \\
 & & \wedge\ (\texttt{unsigned}) \to \texttt{double} \\
 & & \wedge\ (\texttt{doublish}) \to \texttt{double} \\
\texttt{-} & : & (\texttt{int}) \to \texttt{intish} \\
 & & \wedge\ (\texttt{doublish}) \to \texttt{double} \\
\texttt{\~{}} & : & (\texttt{intish}) \to \texttt{signed}
\end{array}
$$

4 Validation

This section describes the `asm.js` validation process, which is essentially a static type system.

4.1 Standard libraries

The JavaScript `Math` API is recognized as a typed standard library; most of its functions are allowed as imports with the same name and are given appropriate function types. These functions must be passed into the import environment under their respective names (e.g., `Math.sin` under the name g`.Math.sin`). Building in support for these standard functions allows optimizing engines to build special support for them—particularly the `imul` operation, which can be implemented with hardware multiplication instructions.

$$\begin{aligned}
M(\texttt{acos}), M(\texttt{asin}), M(\texttt{atan}) &= (\texttt{doublish}) \to \texttt{double} \\
M(\texttt{cos}), M(\texttt{sin}), M(\texttt{tan}) &= (\texttt{doublish}) \to \texttt{double} \\
M(\texttt{ceil}), M(\texttt{floor}) &= (\texttt{doublish}) \to \texttt{double} \\
M(\texttt{exp}), M(\texttt{log}), M(\texttt{sqrt}) &= (\texttt{doublish}) \to \texttt{double} \\
M(\texttt{abs}) &= (\texttt{signed}) \to \texttt{unsigned} \\
&\wedge\ (\texttt{doublish}) \to \texttt{double} \\
M(\texttt{atan2}), M(\texttt{pow}) &= (\texttt{doublish}, \texttt{doublish}) \to \texttt{double} \\
M(\texttt{imul}) &= (\texttt{int}, \texttt{int}) \to \texttt{signed} \\
M(\texttt{random}) &= () \to \texttt{double} \\
M(\texttt{E}) &= \texttt{double} \\
M(\texttt{LN10}), M(\texttt{LN2}), M(\texttt{LOG2E}), M(\texttt{LOG10E}) &= \texttt{double} \\
M(\texttt{PI}) &= \texttt{double} \\
M(\texttt{SQRT1_2}), M(\texttt{SQRT2}) &= \texttt{double}
\end{aligned}$$

4.2 View types

The typed array constructors are also recognized as imports in the import environment, each under its standard name. Calling the various typed arrays constructors on the buffer produces typed array views of various types.

$$\begin{aligned}
V(\texttt{Uint8Array}), V(\texttt{Int8Array}) &= \texttt{view}^{1}_{\texttt{intish}} \\
V(\texttt{Uint16Array}), V(\texttt{Int16Array}) &= \texttt{view}^{2}_{\texttt{intish}} \\
V(\texttt{Uint32Array}), V(\texttt{Int32Array}) &= \texttt{view}^{4}_{\texttt{intish}} \\
V(\texttt{Float32Array}) &= \texttt{view}^{4}_{\texttt{doublish}} \\
V(\texttt{Float64Array}) &= \texttt{view}^{8}_{\texttt{doublish}}
\end{aligned}$$

4.3 Global constants

$$G(\texttt{Infinity}), G(\texttt{NaN}) = \texttt{double}$$

4.4 Annotations

Type annotations are provided in the form of explicit coercions. Variables in `asm.js` are always taken to have the type `double` or `int`, never `signed` or `unsigned`. This is because they are intended to be representable as unboxed 32-bit words in memory or registers, which are agnostic about what sign to interpret the bits with.

$$\begin{array}{rcl} \textit{var-type}(\texttt{+}x), \textit{var-type}(r) & = & \texttt{double} \\ \textit{var-type}(x\texttt{|0}), \textit{var-type}(i) & = & \texttt{int}\ (-2^{31} \leq i < 2^{32}) \end{array}$$

Function return types are determined by explicit coercions in their return statements. Function return types are always explicit about their sign so that they can be exported to external JavaScript.

$$\begin{array}{rcl} \textit{return-type}(\texttt{+}e), \textit{return-type}(r) & = & \texttt{double} \\ \textit{return-type}(e\texttt{|0}), \textit{return-type}(i) & = & \texttt{signed}\ (-2^{31} \leq i < 2^{31}) \\ \textit{return-type}(\epsilon) & = & \texttt{void} \end{array}$$

The type of a function can be extracted by looking at its parameter annotations and return statements.

$\textit{fun-type}(\texttt{function}\ f\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\ \overline{x\ \texttt{=}\ \textit{ann}_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ ss\ \texttt{return}\ [re]\texttt{;}\ \texttt{\}}) = (\overline{\sigma}) \to \tau$
where $\forall i.\textit{var-type}(\textit{ann}_{x_i}) = \sigma_i$
and $\textit{return-type}([re]) = \tau$

$\textit{fun-type}(\texttt{function}\ f\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\ \overline{x\ \texttt{=}\ \textit{ann}_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ ss\ s\ \texttt{\}}) = (\overline{\sigma}) \to \texttt{void}$
where $\forall i.\textit{var-type}(\textit{ann}_{x_i}) = \sigma_i$
and $s \neq \texttt{return}\ [re]\texttt{;}$

$\textit{fun-type}(\texttt{function}\ f\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\ \overline{x\ \texttt{=}\ \textit{ann}_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ \epsilon\ \texttt{\}}) = (\overline{\sigma}) \to \texttt{void}$
where $\forall i.\textit{var-type}(\textit{ann}_{x_i}) = \sigma_i$

The types of import expressions are determined by the standard library metafunctions G, M, and V or by their coercion. A foreign import with no coercion is assumed to be a function.

$$\begin{array}{l} \textit{import-type}(\textit{global}\texttt{.}x) = G(x) \\ \textit{import-type}(\textit{global}\texttt{.Math.}x) = M(x) \\ \textit{import-type}(\texttt{new}\ \textit{global}\texttt{.}x\texttt{(}\textit{buffer}\texttt{)}) = V(y) \\ \textit{import-type}(\textit{foreign}\texttt{.}x\texttt{|0}) = \texttt{signed} \\ \textit{import-type}(\texttt{+}\textit{foreign}\texttt{.}x) = \texttt{double} \\ \textit{import-type}(\textit{foreign}\texttt{.}x) = \texttt{Function} \end{array}$$

4.5 Module validation

Module validation takes an `asm.js` module and constructs a *global environment* Δ, which is used to track the types of all globals: imports, buffer views, global variables, and functions.

$$\begin{array}{rcl} \mu & ::= & \texttt{mut} \mid \texttt{imm} \\ \Delta & ::= & \{\overline{x : \mu\,\gamma}\} \end{array}$$

Validation then uses this environment to check the imports, exports, and function definitions.

$\boxed{\vdash \mathit{mod}\ \mathbf{ok}}$

[T-Module]

$$\frac{\begin{array}{c}\Delta = \{\overline{x : \texttt{imm}\ \mathit{import\text{-}type}(\mathit{imp})}, \overline{g : \texttt{imm}\ \mathit{fun\text{-}type}(\mathit{fn}_g)}, \overline{y : \texttt{mut}\ \mathit{var\text{-}type}(v)}\} \\ \overline{x}, \overline{y}, \overline{g}, [f], [\mathit{global}], [\mathit{foreign}], [\mathit{buffer}]\ \text{distinct} \\ \forall i.\Delta \vdash \mathit{fn}_f\ \mathbf{ok} \qquad \forall i.\Delta \vdash \mathit{exp}\ \mathbf{ok}\end{array}}{\begin{array}{l}\texttt{function}\ [f]\texttt{(}[\mathit{global}[, \mathit{foreign}[, \mathit{buffer}]]]\texttt{)}\ \texttt{\{} \\ \quad \texttt{"use asm";} \\ \quad \overline{\texttt{var}\ \overline{x\ \texttt{=}\ \mathit{imp}}\texttt{;}} \\ \vdash \quad \overline{\mathit{fn}_g} \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{ok} \\ \quad \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}} \\ \quad \mathit{exp} \\ \texttt{\}}\end{array}}$$

For simplicity of the specification, we leave as an assumption that the same *global*, *foreign*, and *buffer* variables are used consistently throughout the module. An actual validator must check this assumption. Similarly, we assume the existence of a single *mask* constant used throughout the module, which a real validator would have to check.

4.6 Function validation

Function validation proceeds in several steps. First, a local environment is constructed with bindings for the parameters and local variables, based on their annotations and initializers, respectively. Next, the body of the function is checked in this environment. Finally, if the function has a non-`void` return type, the body is checked to ensure that all control flow paths definitely return a value (see Section **??**).

$\boxed{\Delta \vdash \mathit{fn}\ \mathbf{ok}}$

[T-Function]

$$\frac{\begin{array}{c}\overline{x}, \overline{y}\ \text{distinct} \qquad \Delta(f) = \texttt{imm}\,(\overline{\sigma}) \to \tau \qquad \overline{\sigma} = \overline{\mathit{var\text{-}type}(\mathit{ann}_x)} \\ \Delta; \{\overline{x : \sigma}, \overline{y : \mathit{var\text{-}type}(v)}\}; \tau \vdash \mathit{ss}\ \mathbf{ok}\end{array}}{\Delta \vdash \texttt{function}\ f\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\ \overline{x\ \texttt{=}\ \mathit{ann}_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ \mathit{ss}\ \texttt{\}}\ \mathbf{ok}}$$

4.7 Export validation

Export validation ensures that all exports are functions.

$\boxed{\Delta \vdash \mathit{exp}\ \mathbf{ok}}$

[T-Singleton]

$$\frac{\Delta(f) = \texttt{imm}\,(\overline{\sigma}) \to \tau}{\Delta \vdash \texttt{return}\ f\texttt{;}\ \mathbf{ok}}$$

[T-Module]

$$\frac{\forall f.\Delta(f) = \texttt{imm}\,(\overline{\sigma}) \to \tau}{\Delta \vdash \texttt{return \{}\ \overline{x\texttt{:}f}\ \texttt{\};}\ \mathbf{ok}}$$

4.8 Statement list validation

$$\boxed{\Delta; \Gamma; \tau \vdash ss \ \mathbf{ok}}$$

[T-Statements]
$$\frac{\forall i. \Delta; \Gamma; \tau \vdash s_i \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \overline{s} \ \mathbf{ok}}$$

4.9 Statement validation

$$\boxed{\Delta; \Gamma; \tau \vdash s \ \mathbf{ok}}$$

[T-Block]
$$\frac{\Delta; \Gamma; \tau \vdash ss \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{\{ } ss \texttt{ \}} \ \mathbf{ok}}$$

[T-ExprStmt]
$$\frac{\Delta; \Gamma \vdash e : \sigma}{\Delta; \Gamma; \tau \vdash e\texttt{;} \ \mathbf{ok}}$$

[T-EmptyStatement]
$$\frac{}{\Delta; \Gamma; \tau \vdash \texttt{;} \ \mathbf{ok}}$$

[T-If]
$$\frac{\Delta; \Gamma \vdash e : \texttt{boolish} \quad \Delta; \Gamma; \tau \vdash s \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{if (}e\texttt{)} \ s \ \mathbf{ok}}$$

[T-IfElse]
$$\frac{\Delta; \Gamma \vdash e : \texttt{boolish} \quad \Delta; \Gamma; \tau \vdash s_1 \ \mathbf{ok} \quad \Delta; \Gamma; \tau \vdash s_2 \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{if (}e\texttt{)} \ s_1 \ \texttt{else} \ s_2 \ \mathbf{ok}}$$

[T-ReturnExpr]
$$\frac{\Delta; \Gamma \vdash re : \tau \quad \mathit{return\text{-}type}(re) = \tau}{\Delta; \Gamma; \tau \vdash \texttt{return} \ re\texttt{;} \ \mathbf{ok}}$$

[T-ReturnVoid]
$$\frac{}{\Delta; \Gamma; \texttt{void} \vdash \texttt{return;} \ \mathbf{ok}}$$

[T-While]
$$\frac{\Delta; \Gamma \vdash e : \texttt{int} \quad \Delta; \Gamma; \tau \vdash s \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{while (}e\texttt{)} \ s \ \mathbf{ok}}$$

[T-DoWhile]
$$\frac{\Delta; \Gamma; \tau \vdash s \ \mathbf{ok} \quad \Delta; \Gamma \vdash e : \texttt{int}}{\Delta; \Gamma; \tau \vdash \texttt{do} \ s \ \texttt{while (}e\texttt{);} \ \mathbf{ok}}$$

[T-For]
$$\frac{[\Delta; \Gamma \vdash e_1 : \sigma_1] \quad [\Delta; \Gamma \vdash e_2 : \texttt{int}] \quad [\Delta; \Gamma \vdash e_3 : \sigma_3] \quad \Delta; \Gamma; \tau \vdash s \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{for (}[e_1]\texttt{;} \ [e_2]\texttt{;} \ [e_3]\texttt{)} \ s \ \mathbf{ok}}$$

[T-Break]
$$\frac{}{\Delta; \Gamma; \tau \vdash \texttt{break} \ [lab]\texttt{;} \ \mathbf{ok}}$$

[T-Continue]
$$\frac{}{\Delta; \Gamma; \tau \vdash \texttt{continue} \ [lab]\texttt{;} \ \mathbf{ok}}$$

[T-Label]
$$\frac{\Delta; \Gamma; \tau \vdash s \ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash lab\texttt{:} \ s \ \mathbf{ok}}$$

[T-Switch]
$$\frac{\Delta; \Gamma \vdash e : \sigma \quad \sigma \in \{\texttt{signed}, \texttt{unsigned}\} \quad \forall i. \Delta; \Gamma \vdash v_i : \sigma \quad \forall i. \Delta; \Gamma; \tau \vdash ss_i \ \mathbf{ok} \quad [\Delta; \Gamma; \tau \vdash ss \ \mathbf{ok}]}{\Delta; \Gamma; \tau \vdash \texttt{switch (}e\texttt{) \{} \ \overline{\texttt{case} \ v_i\texttt{:} \ ss_i} \ [\texttt{default:} \ ss] \ \texttt{\}} \ \mathbf{ok}}$$

4.10 Case validation

$\boxed{\Delta; \Gamma; \tau \vdash cd\ \mathbf{ok}}$

[T-Case]
$$\frac{\Delta; \Gamma; \tau \vdash ss\ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{case}\ v\texttt{:}\ ss\ \mathbf{ok}}$$

[T-Default]
$$\frac{\Delta; \Gamma; \tau \vdash ss\ \mathbf{ok}}{\Delta; \Gamma; \tau \vdash \texttt{default:}\ ss\ \mathbf{ok}}$$

4.11 Expression validation

$$(\Delta \cdot \Gamma)(x) = \begin{cases} \Gamma(x) & \text{if } x \in dom(\Gamma) \\ \gamma & \text{if } \Delta(x) = \mu\,\gamma \end{cases}$$

Expression validation $\boxed{\Delta; \Gamma \vdash e : \tau}$

[T-Signed]
$$\frac{-2^{31} \le i < 0}{\Delta; \Gamma \vdash i : \texttt{signed}}$$

[T-Fixnum]
$$\frac{0 \le i < 2^{31}}{\Delta; \Gamma \vdash i : \texttt{fixnum}}$$

[T-Unsigned]
$$\frac{2^{31} \le i < 2^{32}}{\Delta; \Gamma \vdash i : \texttt{unsigned}}$$

[T-Double]
$$\frac{}{\Delta; \Gamma \vdash r : \texttt{double}}$$

[T-VarRef]
$$\frac{(\Delta \cdot \Gamma)(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

[T-SetLocal]
$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau <: \Gamma(x)}{\Delta; \Gamma \vdash x\ \texttt{=}\ e : \tau}$$

[T-SetGlobal]
$$\frac{x \notin dom(\Gamma) \qquad \Delta(x) = \texttt{mut}\,\sigma \qquad \Delta; \Gamma \vdash e : \tau \qquad \tau <: \sigma}{\Delta; \Gamma \vdash x\ \texttt{=}\ e : \tau}$$

[T-LoadImm]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \qquad i \equiv 0 \bmod n \qquad 0 \le i \le \mathit{mask} \qquad \Delta; \Gamma \vdash e : \texttt{intish}}{\Delta; \Gamma \vdash x\texttt{[}i\texttt{]} : \tau}$$

[T-StoreImm]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \qquad i \equiv 0 \bmod n \qquad 0 \le i \le \mathit{mask} \qquad \Delta; \Gamma \vdash e_1 : \texttt{intish} \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash x\texttt{[}i\texttt{]}\ \texttt{=}\ e_2 : \tau}$$

[T-LoadByte]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^1_{\texttt{intish}} \qquad \Delta; \Gamma \vdash e : \texttt{intish}}{\Delta; \Gamma \vdash x\texttt{[}e\ \texttt{\&}\ \mathit{mask}\texttt{]} : \tau}$$

[T-StoreByte]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^1_{\texttt{intish}} \qquad \Delta; \Gamma \vdash e_1 : \texttt{intish} \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash x\texttt{[}e_1\ \texttt{\&}\ \mathit{mask}\texttt{]}\ \texttt{=}\ e_2 : \tau}$$

[T-Load]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \qquad \mathit{shift} = \log_2(n) \qquad n > 1 \qquad \Delta; \Gamma \vdash e : \texttt{intish}}{\Delta; \Gamma \vdash x\texttt{[(}e\ \texttt{\&}\ \mathit{mask}\texttt{)}\ \texttt{>>}\ \mathit{shift}\texttt{]} : \tau}$$

[T-Store]
$$\frac{(\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \qquad \mathit{shift} = \log_2(n) \qquad n > 1 \qquad \Delta; \Gamma \vdash e_1 : \texttt{intish} \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash x\texttt{[(}e_1\ \texttt{\&}\ \mathit{mask}\texttt{)}\ \texttt{>>}\ \mathit{shift}\texttt{]}\ \texttt{=}\ e_2 : \tau}$$

Expression validation (cont'd) $\boxed{\Delta;\Gamma \vdash e : \tau}$

[T-FunCall]
$$\frac{(\Delta \cdot \Gamma)(f) = \_ \wedge (\overline{\sigma}) \to \tau \wedge \_ \qquad \forall i.\Delta;\Gamma \vdash e_i : \sigma_i}{\Delta;\Gamma \vdash f\,(\overline{e}) : \tau}$$

[T-FFICall]
$$\frac{(\Delta \cdot \Gamma)(f) = \texttt{Function} \qquad \forall i.\Delta;\Gamma \vdash e_i : \texttt{extern}}{\Delta;\Gamma \vdash f\,(\overline{e}) : \texttt{unknown}}$$

[T-Unop]
$$\frac{\mathit{unop} : \_ \wedge (\sigma) \to \tau \wedge \_ \qquad \Delta;\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \mathit{unop}\ e : \tau}$$

[T-Binop]
$$\frac{\mathit{binop} : \_ \wedge (\sigma_1, \sigma_2) \to \tau \wedge \_ \qquad \Delta;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta;\Gamma \vdash e_2 : \sigma_2}{\Delta;\Gamma \vdash e_1\ \mathit{binop}\ e_2 : \tau}$$

[T-Multiary]
$$\frac{\forall i < n.\oplus_i \in \{+,-\} \qquad n \leq 2^{20} \qquad \forall i \leq n.\Delta;\Gamma \vdash e_i : \texttt{int}}{\Delta;\Gamma \vdash e_1 \oplus_1 \ldots \oplus_{n-1} e_n : \texttt{intish}}$$

[T-Cond]
$$\frac{\tau \in \{\texttt{int}, \texttt{double}\} \qquad \Delta;\Gamma \vdash e_1 : \texttt{int} \qquad \Delta;\Gamma \vdash e_2 : \tau \qquad \Delta;\Gamma \vdash e_3 : \tau}{\Delta;\Gamma \vdash e_1\,\texttt{?}\,e_2\,\texttt{:}e_3 : \tau}$$

[T-Paren]
$$\frac{\forall i \leq n.\Delta;\Gamma \vdash e_i : \tau_i}{\Delta;\Gamma \vdash (\overline{e}) : \tau_n}$$

[T-Sub]
$$\frac{\Delta;\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Delta;\Gamma \vdash e : \tau}$$

[T-Cast]
$$\frac{\Delta;\Gamma \vdash e : \texttt{double}}{\Delta;\Gamma \vdash \texttt{~~}e : \texttt{signed}}$$