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

Dave Herman, Luke Wagner, and Alon Zakai

November 19, 2012

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 two optional parameters: an *environment*, containing imported functions and constants from external JavaScript (serving as one side of a kind of foreign-function interface), 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(env, buffer) {
    "use asm";

    var HEAP_I32 = new env.Int32Array(buffer);
    var HEAP_F64 = new env.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(env, buffer) {
    "use asm";
    // ...
    return { f: foo, g: bar };
}

var env = {
    Int32Array: Int32Array,
    Uint32Array: Uint32Array,
    // ...
};
var buf = new ArrayBuffer(0x100000);

// link the module
var m = mod(env, buf);
```

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.

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 b, c, f, g, x, y, and z are used to range over JavaScript identifiers.

2.1 Modules

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

$$\mathit{mod} ::= \texttt{function }[g]\texttt{(}[c[, b]]\texttt{) \{ "use asm"; }\ \overline{\mathit{imp}_x}\ \overline{\mathit{fn}_f}\ \overline{\texttt{var }\overline{y\texttt{ = }v}\texttt{;}}\ \mathit{exp}\ \texttt{\}}$$

The syntax consists of:

1. up to two optional parameters;
2. a `"use asm";` prologue directive;
3. a sequence of import statements;
4. a sequence of function declarations;
5. a sequence of global variable declarations; and
6. a single export statement.

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

$$\begin{array}{rcl}
\mathit{imp}_x & ::= & \texttt{var } x \texttt{ = } c\texttt{.}y\texttt{;} \\
& | & \texttt{var } x \texttt{ = } c\texttt{.}y\texttt{|0;} \\
& | & \texttt{var } x \texttt{ = +}c\texttt{.}y\texttt{;} \\
& | & \texttt{var } x \texttt{ = new } c\texttt{.}y\texttt{(}b\texttt{);}
\end{array}$$

A function declaration has the following syntax:

$$fn_f ::= \texttt{function}\ f(\overline{x})\ \texttt{\{}\ \overline{x\ \texttt{=}\ ann_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ ss\ \texttt{\}}$$

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:

$$ann_x ::= x\texttt{|0} \mid \texttt{+}x$$

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

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

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:

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

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, conditional expressions, and sequence expressions:

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

Literals are either doubles or integers:

$$v ::= r \mid n$$

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.

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

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.

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

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

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

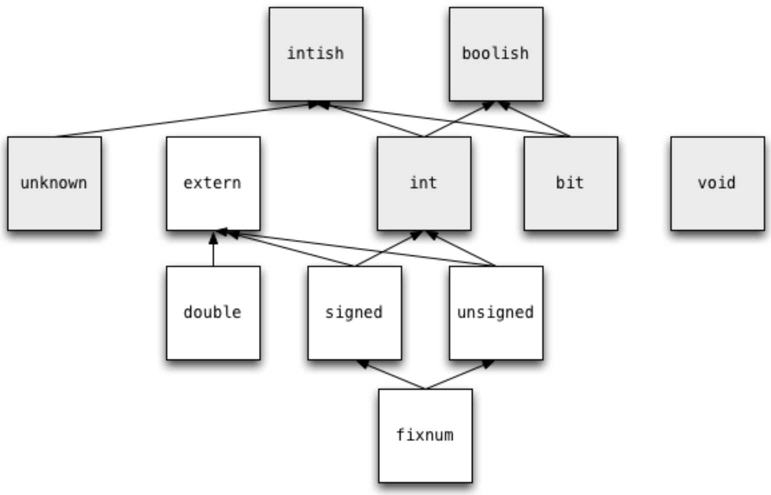

Figure 1: The hierarchy of expression types.

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.

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`.

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 `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.

The `bit` type

The conditional operators produce boolean values in JavaScript. Booleans are interconvertible to the JavaScript integer values `0` and `1` via integer coercion and back via boolean coercion. In order to allow optimizing engines to represent these as untagged 32-bit integers, we disallow the `bit` type from escaping to external JavaScript. The only contexts where they can be used are in conditionals (such as the test expression of an `if` statement), or in coercions to integer.

The `boolish` type

Every type in JavaScript can be implicitly converted to a boolean for the sake of evaluating a conditional expression or statement. This coercion is allowed in `asm.js` for the `bit` type as well as the integer types, but not for the `double` type. This ensures that an optimizing JavaScript engine only ever performs a conditional based on an unboxed 32-bit integer value, avoiding the need for a more expensive conversion.

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 Environment types

Validation tracks the types not only of expressions and local variables but also global variables, FFI imports, and functions. These include typed arrays, the special `Math.imul` value, FFI functions, and `asm.js` functions:

$$\rho ::= \tau \mid \mathtt{view}^n_\tau \mid \mathtt{imul} \mid \mathtt{function} \mid (\overline{\sigma}) \to \tau$$

The type of a typed array tracks the number of bits per element and the elements' value type (`int` or `double`). The `imul` and `function` types are straightforward. The type of a function define within the `asm.js` module includes the types of its parameters and its return type.

3.3 Operator types

$$\omega ::= ((\overline{\sigma}) \to \tau) \wedge \ldots \wedge ((\overline{\sigma'}) \to \tau')$$

4 Type rules

$$\begin{array}{rcl}
\mathit{var\text{-}type}(\texttt{+}x), \mathit{var\text{-}type}(r) & = & \texttt{double} \\
\mathit{var\text{-}type}(x\texttt{|0}), \mathit{var\text{-}type}(n) & = & \texttt{int}\ (-2^{31} \leq n < 2^{32}) \\
\mathit{return\text{-}type}(\texttt{+}e), \mathit{return\text{-}type}(r) & = & \texttt{double} \\
\mathit{return\text{-}type}(e\texttt{|0}), \mathit{return\text{-}type}(n) & = & \texttt{signed}\ (-2^{31} \leq n < 2^{31}) \\
\mathit{return\text{-}type}(\epsilon) & = & \texttt{void}
\end{array}$$

$$\begin{array}{rcl}
M(\texttt{imul}) & : & \texttt{imul} \\
M(\texttt{ceil}), M(\texttt{sin}), M(\texttt{cos}) & : & (\texttt{double}) \to \texttt{double}
\end{array}$$

$$\begin{array}{rcl}
A(\texttt{Uint8Array}), A(\texttt{Int8Array}) & = & \texttt{view}^{8}_{\texttt{int}} \\
A(\texttt{Uint16Array}), A(\texttt{Int16Array}) & = & \texttt{view}^{16}_{\texttt{int}} \\
A(\texttt{Uint32Array}), A(\texttt{Int32Array}) & = & \texttt{view}^{32}_{\texttt{int}} \\
A(\texttt{Float32Array}) & = & \texttt{view}^{32}_{\texttt{double}} \\
A(\texttt{Float64Array}) & = & \texttt{view}^{64}_{\texttt{double}}
\end{array}$$

$$\begin{array}{rcl}
\texttt{+}, \texttt{-} & : & \ (\texttt{double}, \texttt{double}) \to \texttt{double} \\
 & & \wedge\ (\texttt{int}, \texttt{int}) \to \texttt{intish} \\
\texttt{*} & : & \ (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\texttt{/}, \texttt{\%} & : & \ (\texttt{double}, \texttt{double}) \to \texttt{double} \\
 & & \wedge\ (\texttt{signed}, \texttt{signed}) \to \texttt{intish} \\
 & & \wedge\ (\texttt{unsigned}, \texttt{unsigned}) \to \texttt{intish} \\
 & & \\
\texttt{|}, \texttt{\&}, \texttt{\^{}}, \texttt{<<}, \texttt{>>} & : & \ (\texttt{intish}, \texttt{intish}) \to \texttt{signed} \\
\texttt{>>>} & : & \ (\texttt{intish}, \texttt{intish}) \to \texttt{unsigned} \\
 & & \\
\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=} & : & \ (\texttt{signed}, \texttt{signed}) \to \texttt{bit} \\
 & & \wedge\ (\texttt{unsigned}, \texttt{unsigned}) \to \texttt{bit} \\
 & & \wedge\ (\texttt{double}, \texttt{double}) \to \texttt{bit} \\
 & & \\
\texttt{+} & : & \ (\texttt{intish}) \to \texttt{double} \\
\texttt{\~{}} & : & \ (\texttt{intish}) \to \texttt{signed} \\
\texttt{!} & : & \ (\texttt{boolish}) \to \texttt{bit}
\end{array}$$

$$\begin{array}{rcl}
\Delta & ::= & \{\overline{x : \rho}\} \\
\Gamma & ::= & \{\overline{x : \tau}\}
\end{array}$$

$$\mathit{fun\text{-}type}(\texttt{function}\ f\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\ \overline{x\ \texttt{=}\ ann_x\texttt{;}}\ \overline{\texttt{var}\ \overline{y\ \texttt{=}\ v}\texttt{;}}\ ss\ \texttt{\}}) = (\overline{\sigma}) \to \tau$$

where $\forall i.\mathit{var\text{-}type}(ann_{x_i}) = \sigma_i$

and $\forall \texttt{return}\ [re]\texttt{;} \in ss.\mathit{return\text{-}type}([re]) = \tau$

$$
\begin{array}{rcl}
\mathit{breaks}(\overline{s}) & = & \bigcup_i \mathit{breaks}(s_i) \\
\mathit{breaks}(\texttt{\{}\ \mathit{ss}\ \texttt{\}}) & = & \mathit{breaks}(\mathit{ss}) \\
\mathit{breaks}(\texttt{if (}e\texttt{) } s) & = & \mathit{breaks}(s) \\
\mathit{breaks}(\texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2) & = & \mathit{breaks}(s_1) \cup \mathit{breaks}(s_2) \\
\mathit{breaks}(\texttt{while (}e\texttt{) } s) & = & \mathit{breaks}(s) - \{\epsilon\} \\
\mathit{breaks}(\texttt{do } s \texttt{ while (}e\texttt{);}) & = & \mathit{breaks}(s) - \{\epsilon\} \\
\mathit{breaks}(\texttt{for (}[e_1]\texttt{; }[e_2]\texttt{; }[e_3]\texttt{) } s) & = & \mathit{breaks}(s) - \{\epsilon\} \\
\mathit{breaks}(\texttt{break;}) & = & \{\epsilon\} \\
\mathit{breaks}(\texttt{break } \mathit{lab}\texttt{;}) & = & \{\mathit{lab}\} \\
\mathit{breaks}(\mathit{lab}\texttt{:}\ s) & = & \mathit{breaks}(s) - \{\mathit{lab}\} \\
\mathit{breaks}(\texttt{switch (}e\texttt{) \{ } \overline{\mathit{cd}} \texttt{ \}}) & = & \bigcup_i \mathit{breaks}(\mathit{cd}_i) - \{\epsilon\} \\
\mathit{breaks}(s)\ \text{(otherwise)} & = & \emptyset \\
\mathit{breaks}(\texttt{case } v\texttt{:}\ \mathit{ss}) & = & \mathit{breaks}(\mathit{ss}) \\
\mathit{breaks}(\texttt{default:}\ \mathit{ss}) & = & \mathit{breaks}(\mathit{ss})
\end{array}
$$

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

[T-Program]

$$
\frac{\begin{array}{c}\overline{x}, \overline{y}, \overline{f}, [g], [e], [b] \text{ distinct} \qquad \Delta = \{\overline{f : \mathit{fun\text{-}type}(\mathit{fn}_f)}, \overline{y : \mathit{var\text{-}type}(v)}\} \\ \forall i.[e];[b];\Delta \vdash \mathit{imp}_x\ \mathbf{ok} \qquad \forall i.\Delta \vdash \mathit{fn}_f\ \mathbf{ok} \qquad \forall i.\Delta \vdash \mathit{exp}\ \mathbf{ok}\end{array}}{\vdash \texttt{function } [g]\texttt{(}[e[,b]]\texttt{) \{ "use asm"; } \overline{\mathit{imp}_x}\ \overline{\mathit{fn}_f}\ \overline{\texttt{var } \overline{y \texttt{ = } v}\texttt{;}}\ \mathit{exp} \texttt{ \}}\ \mathbf{ok}}
$$

Import checking $\boxed{[e];[b];\Delta \vdash \mathit{imp}\ \mathbf{ok}}$

[T-ImportStd]

$$
\frac{\Delta(x) = M(y)}{e;[b];\Delta \vdash \texttt{var } x \texttt{ = } e\texttt{.}y\texttt{;}\ \mathbf{ok}}
$$

[T-ImportFFI]

$$
\frac{y \notin \mathit{dom}(M), \mathit{dom}(A) \qquad \Delta(x) = \texttt{function}}{e;[b];\Delta \vdash \texttt{var } x \texttt{ = } e\texttt{.}y\texttt{;}\ \mathbf{ok}}
$$

[T-View]

$$
\frac{\Delta(x) = \texttt{view}^n_{A(y)}}{e;b;\Delta \vdash \texttt{var } x \texttt{ = new } e\texttt{.}y(b)\texttt{;}\ \mathbf{ok}}
$$

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

[T-Function]

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

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

[T-Singleton]

$$
\frac{\Delta(f) = (\overline{\sigma}) \to \tau \qquad \tau <: \texttt{extern}}{\Delta \vdash \texttt{return } f\texttt{;}\ \mathbf{ok}}
$$

[T-Module]

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

$returns(\overline{s})$

  if $returns(s_m) \land \forall i < m. breaks(s_m) = \emptyset$ for some m

$returns(\texttt{\{}\ ss\ \texttt{\}})$

  if $returns(ss)$

$returns(\texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2)$

  if $returns(s_1) \land returns(s_2)$

$returns(\texttt{do}\ s\ \texttt{while (}e\texttt{);})$

  if $returns(s)$

$returns(\texttt{switch (}e\texttt{) \{}\ \overline{cd}\ \texttt{\}})$

  if $returns(cd_n) \land \forall i. breaks(cd_i) = \emptyset$

$returns(\texttt{case}\ v\texttt{:}\ ss)$

  if $returns(ss)$

$returns(\texttt{default:}\ ss)$

  if $returns(ss)$

Statement list checking $\boxed{\Delta; \Gamma \vdash ss \textbf{ ok}}$

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

Statement checking $\boxed{\Delta; \Gamma \vdash s \textbf{ ok}}$

[T-BLOCK]
$$\frac{\Delta; \Gamma \vdash ss \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{\{ } ss \texttt{ \}} \textbf{ ok}}$$

[T-EXPRSTMT]
$$\frac{\Delta; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash e\texttt{;} \textbf{ ok}}$$

[T-EMPTYSTATEMENT]
$$\frac{}{\Delta; \Gamma \vdash \texttt{;} \textbf{ ok}}$$

[T-IF]
$$\frac{\Delta; \Gamma \vdash e : \texttt{boolish} \qquad \Delta; \Gamma \vdash s \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{if (}e\texttt{) } s \textbf{ ok}}$$

[T-IFELSE]
$$\frac{\Delta; \Gamma \vdash e : \texttt{boolish} \qquad \Delta; \Gamma \vdash s_1 \textbf{ ok} \qquad \Delta; \Gamma \vdash s_2 \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2 \textbf{ ok}}$$

[T-RETURNEXPR]
$$\frac{\Delta; \Gamma \vdash re : \tau \qquad \textit{return-type}(re) = \tau}{\Delta; \Gamma \vdash \texttt{return } re\texttt{;} \textbf{ ok}}$$

[T-RETURNVOID]
$$\frac{}{\Delta; \Gamma \vdash \texttt{return;} \textbf{ ok}}$$

[T-WHILE]
$$\frac{\Delta; \Gamma \vdash e : \texttt{boolish} \qquad \Delta; \Gamma \vdash s \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{while (}e\texttt{) } s \textbf{ ok}}$$

[T-DOWHILE]
$$\frac{\Delta; \Gamma \vdash s \textbf{ ok} \qquad \Delta; \Gamma \vdash e : \texttt{boolish}}{\Delta; \Gamma \vdash \texttt{do } s \texttt{ while (}e\texttt{);} \textbf{ ok}}$$

[T-FOR]
$$\frac{[\Delta; \Gamma \vdash e_1 : \sigma_1] \qquad [\Delta; \Gamma \vdash e_2 : \texttt{boolish}] \qquad [\Delta; \Gamma \vdash e_3 : \sigma_3] \qquad \Delta; \Gamma \vdash s \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{for (}[e_1]\texttt{; }[e_2]\texttt{; }[e_3]\texttt{) } s \textbf{ ok}}$$

[T-BREAK]
$$\frac{}{\Delta; \Gamma \vdash \texttt{break } [lab]\texttt{;} \textbf{ ok}}$$

[T-CONTINUE]
$$\frac{}{\Delta; \Gamma \vdash \texttt{continue } [lab]\texttt{;} \textbf{ ok}}$$

[T-LABEL]
$$\frac{\Delta; \Gamma \vdash s \textbf{ ok}}{\Delta; \Gamma \vdash lab\texttt{:}\, s \textbf{ ok}}$$

[T-SWITCH]
$$\frac{\Delta; \Gamma \vdash e : \sigma \qquad \sigma <: \texttt{extern} \qquad \forall i. \Delta; \Gamma \vdash v_i : \sigma \qquad \forall i. \Delta; \Gamma \vdash ss_i \textbf{ ok} \qquad [\Delta; \Gamma \vdash ss \textbf{ ok}]}{\Delta; \Gamma \vdash \texttt{switch (}e\texttt{) \{ } \overline{\texttt{case } v_i \texttt{:}\, ss_i}\, [\texttt{default:}\, ss] \texttt{ \}} \textbf{ ok}}$$

Case checking $\boxed{\Delta; \Gamma \vdash cd \textbf{ ok}}$

[T-CASE]
$$\frac{\Delta; \Gamma \vdash ss \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{case } v\texttt{:}\, ss \textbf{ ok}}$$

[T-DEFAULT]
$$\frac{\Delta; \Gamma \vdash ss \textbf{ ok}}{\Delta; \Gamma \vdash \texttt{default:}\, ss \textbf{ ok}}$$

$$(\Delta \cdot \Gamma)(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma) \\ \Delta(x) & \text{otherwise} \end{cases}$$

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

$$\frac{-2^{31} \le n < 0}{\Delta; \Gamma \vdash n : \texttt{signed}}\ [\textsc{T-Signed}] \qquad \frac{0 \le n < 2^{31}}{\Delta; \Gamma \vdash n : \texttt{fixnum}}\ [\textsc{T-Fixnum}] \qquad \frac{2^{31} \le n < 2^{32}}{\Delta; \Gamma \vdash n : \texttt{unsigned}}\ [\textsc{T-Unsigned}]$$

$$\frac{}{\Delta; \Gamma \vdash r : \texttt{double}}\ [\textsc{T-Double}] \qquad \frac{(\Delta \cdot \Gamma)(x) = \tau}{\Delta; \Gamma \vdash x : \tau}\ [\textsc{T-VarRef}] \qquad \frac{\Delta; \Gamma \vdash e : \tau \qquad \tau <: (\Delta \cdot \Gamma)(x)}{\Delta; \Gamma \vdash x\ \texttt{=}\ e : \tau}\ [\textsc{T-Assign}]$$

$$\frac{\begin{array}{c} m = 2^k - 1 \qquad (\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \\ \Delta; \Gamma \vdash e : \texttt{intish} \end{array}}{\Delta; \Gamma \vdash x\texttt{[(}e\ \texttt{\&}\ m\texttt{)}\ \texttt{>>}\ n/8\texttt{]} : \tau}\ [\textsc{T-Load}] \qquad \frac{\begin{array}{c} m = 2^k - 1 \qquad (\Delta \cdot \Gamma)(x) = \texttt{view}^n_\tau \\ \Delta; \Gamma \vdash e_1 : \texttt{intish} \qquad \Delta; \Gamma \vdash e_2 : \tau \end{array}}{\Delta; \Gamma \vdash x\texttt{[(}e_1\ \texttt{\&}\ m\texttt{)}\ \texttt{>>}\ n/8\texttt{]}\ \texttt{=}\ e_2 : \tau}\ [\textsc{T-Store}]$$

$$\frac{\begin{array}{c} (\Delta \cdot \Gamma)(f) = \texttt{imul} \\ \forall i. \Delta; \Gamma \vdash e_i : \texttt{intish} \end{array}}{\Delta; \Gamma \vdash f\texttt{(}e_1, e_2\texttt{)} : \texttt{signed}}\ [\textsc{T-IMul}] \qquad \frac{\begin{array}{c} (\Delta \cdot \Gamma)(f) = (\overline{\sigma}) \to \tau \\ \forall i. \Delta; \Gamma \vdash e_i : \sigma_i \end{array}}{\Delta; \Gamma \vdash f\texttt{(}\overline{e}\texttt{)} : \tau}\ [\textsc{T-FunCall}] \qquad \frac{\begin{array}{c} (\Delta \cdot \Gamma)(f) = \texttt{function} \\ \forall i. \Delta; \Gamma \vdash e_i : \texttt{extern} \end{array}}{\Delta; \Gamma \vdash f\texttt{(}\overline{e}\texttt{)} : \texttt{unknown}}\ [\textsc{T-FFI}]$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e_1 : \texttt{boolish} \\ \Delta; \Gamma \vdash e_2 : \tau \qquad \Delta; \Gamma \vdash e_3 : \tau \end{array}}{\Delta; \Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}\ [\textsc{T-Conditional}] \qquad \frac{\forall i \le n. \Delta; \Gamma \vdash e_i : \tau_i}{\Delta; \Gamma \vdash \texttt{(}\overline{e}\texttt{)} : \tau_n}\ [\textsc{T-Paren}]$$

$$\frac{\mathit{unop} : \_ \wedge (\sigma) \to \tau \wedge \_ \qquad \Delta; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \mathit{unop}\ e : \tau}\ [\textsc{T-Unop}] \qquad \frac{\begin{array}{c} \mathit{binop} : \_ \wedge (\sigma_1, \sigma_2) \to \tau \wedge \_ \\ \Delta; \Gamma \vdash e_1 : \sigma_1 \qquad \Delta; \Gamma \vdash e_2 : \sigma_2 \end{array}}{\Delta; \Gamma \vdash e_1\ \mathit{binop}\ e_2 : \tau}\ [\textsc{T-Binop}]$$

$$\frac{\Delta; \Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Delta; \Gamma \vdash e : \tau}\ [\textsc{T-Sub}] \qquad \frac{\Delta; \Gamma \vdash e : \texttt{double}}{\Delta; \Gamma \vdash \texttt{\~{}\~{}}e : \texttt{signed}}\ [\textsc{T-Cast}]$$