A Little on V8 and WebAssembly

Ben L. Titzer
Virtual Machines Summer School 2016
2016-06-01
Agenda

● What makes JavaScript unique and challenging?
● What makes V8 unique and challenging?
● What the heck is WebAssembly and why?
We all love JavaScript
What makes JavaScript unique and interesting?

- JavaScript is the language of the Web
- Scripting language: programs presented in source form
- “Classically slow” language
- Prototype-based object model
- Functional features with closures
- Untyped: variables and properties do not have types, values do
- A smattering of oddball features
  - Weird scoping rules
  - `eval`
  - `with` scopes
  - Proxies
  - Rest parameters
  - Default parameters
  - Generators
  - Undetectables
  - Holey arrays
  - Arguments object
Challenge: programs presented in source form

- Parsing has to be fast
- Source code is slower for machines to parse
  - Source code parser: 1-10MB/s
  - Binary format like bytecode: 100MB/s
- New language features all the time
  - All features supported by all virtual machines
var x = new SubClass("mine", 100);

function BaseClass(name) {
  this.name = name;
}

function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}

BaseClass.prototype.print = function() {
  print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
Challenge: prototype-based object model

```javascript
var x = new SubClass("mine", 100);

function BaseClass(name) {
    this.name = name;
}

function SubClass(name, data) {
    BaseClass.call(this, name);
    this.data = data;
}

BaseClass.prototype.print = function() {
    print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

- Objects instantiated by “new Function()” syntax
- Methods installed on the “prototype” of an object
- Prototypes chain together to emulate inheritance
Challenge: prototype-based object model

var x = new SubClass("mine", 100);

function BaseClass(name) {
  this.name = name;
}
function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}
BaseClass.prototype.print = function() {
  print(this.name);
}
SubClass.prototype.__proto__ = BaseClass.prototype;

- Objects instantiated by "new Function()" syntax
- **Methods installed on the "prototype" of an object**
- Prototypes chain together to emulate inheritance
Challenge: prototype-based object model

```javascript
var x = new SubClass("mine", 100);

function BaseClass(name) {
    this.name = name;
}

function SubClass(name, data) {
    BaseClass.call(this, name);
    this.data = data;
}

BaseClass.prototype.print = function() {
    print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

- Objects instantiated by “new Function()” syntax
- Methods installed on the “prototype” of an object
- Prototypes chain together to emulate inheritance
function Counter(name) {
  var count = 0;
  return {
    inc: function() { count++; },
    get: function() { return count; },
    print: function() { print(name + ":" + count); }
  }
}

var x = new Counter();

var before = x.get();
x.inc();
x.inc();
x.print();
function Counter(name) {
  var count = 0;
  return {
    inc: function() { count++; },
    get: function() { return count; },
    print: function() { print(name + ":" + count); }
  }
}

var x = new Counter();

var before = x.get();
x.inc();
x.inc();
x.print();

- Closures over locals, even mutable locals
- Object literals allow grouping multiple closures into a "mini-object"
Challenge: untyped variables and operations

```
function add(a, b) {
    return a + b;
}
add(1, 2);
add("foo", 1);
add(1, "foo");
add({foo: ""}, 1);
add("hello", {toString: () => "me"});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values
Challenge: untyped variables and operations

```javascript
function add(a, b) {
    return a + b;
}
add(1, 2);
add("foo", 1);
add(1, "foo");
add({foo: ""}, 1);
add("hello", {toString: () => "me"});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values
Challenge: untyped variables and operations

function add(a, b) {
    return a + b;
}
add(1, 2);
add("foo", 1);
add(1, "foo");
add({foo: ""}, 1);
add("hello", {toString: () => "me"});
add(1.01, 3.03);

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values
12.7.3.1 Runtime Semantics: Evaluation

**operator +**

AdditiveExpression : AdditiveExpression operator + MultiplicativeExpression

1. Let `lval` be the result of evaluating `AdditiveExpression`.
2. Let `lvl` be `GetValue(lval)`.
3. ReturnIfAbrupt(lvl).
4. Let `rval` be the result of evaluating `MultiplicativeExpression`.
5. Let `rval` be `GetValue(rval)`.
6. ReturnIfAbrupt(rval).
7. Let `lprim` be `ToPrimitive(lval)`.
8. ReturnIfAbrupt(lprim).
9. Let `rprim` be `ToPrimitive(rval)`.
10. ReturnIfAbrupt(rprim).
11. If `Type(lprim)` is `String` or `Type(rprim)` is `String`, then
    a. Let `bar` be `ToString(lprim)`.
    b. ReturnIfAbrupt(bar).
    c. Let `rstr` be `ToString(rprim)`.
    d. ReturnIfAbrupt(rstr).
    e. Return the `String` that is the result of concatenating `bar` and `rstr`.
12. Let `inum` be `ToNumber(lprim)`.
13. ReturnIfAbrupt(inum).
14. Let `rnum` be `ToNumber(rprim)`.
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the `addition` operation to `inum` and `rnum`. See the Note below 12.7.5.

**NOTE 1** No hint is provided in the calls to `ToPrimitive` in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint `Number` were given; Date objects handle the absence of a hint as if the hint `String` were given. Exotic objects may handle the absence of a hint in some other manner.

**NOTE 2** Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.
Glance at Semantics: +

12.7.3.1 Runtime Semantics: Evaluation

**Operator +**

1. Let `left` be the result of evaluating `AdditiveExpression`.
2. Let `right` be the result of evaluating `AdditiveExpression`.
3. Return `left` + `right`.

**ToPrimitive**

The abstract operator `ToPrimitive` is used to convert values to their primitive values. The `ToPrimitive` operator is defined as follows:

**Table 9 — ToPrimitive Conversions**

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td><code>null</code></td>
</tr>
<tr>
<td>Boolean</td>
<td><code>true</code> or <code>false</code></td>
</tr>
<tr>
<td>Number</td>
<td><code>Number</code></td>
</tr>
<tr>
<td>String</td>
<td><code>String</code></td>
</tr>
<tr>
<td>Symbol</td>
<td><code>Symbol</code></td>
</tr>
<tr>
<td>Object</td>
<td><code>Object</code></td>
</tr>
</tbody>
</table>

When `ToPrimitive` is called with arguments `O` and `hint`, the following steps are taken:

1. **hint**
2. **Object**
3. **Symbol**
4. **String**

**NOTE 1**
No hint is provided in the calls to `ToPrimitive` in steps 7 and 8. All standard objects except Date objects handle the absence of a hint as if the hint Number were given. Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

**NOTE 2**
Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.3.1), by using the logical-or operation instead of the logical-and operation.
Glance at Semantics: +

7.1.12.4.1.1 Barrier Semantics Evaluation

This subsection discusses the evaluation of the Barrier operator. The evaluation process involves checking the semantics of the operands and determining the result based on the type of the operands. The subsection includes examples and illustrations to help understand the evaluation process.

---

7.1.12.4.1.2 Toplevel (argument)

The toplevel operator Toplevel returns the result of evaluating the expression in parentheses. This operator is used to evaluate expressions within a function, providing a way to evaluate expressions in a controlled environment.

---

7.1.12.4.1.3 ToString

The toString operator converts a value to a string representation. This operator is used to convert various data types, such as numbers, objects, and arrays, into a string format.

---

7.1.12.4.1.4 ToPrimitive

The toPrimitive operator converts a value into a primitive type. This operator is used to convert objects and values into their most basic form, ensuring compatibility and consistency in data manipulation.

---

Example:

- ToPrimitive can be used to ensure that a function receives arguments in a consistent format, which is crucial for functions that expect specific types of inputs.

---

7.1.12.4.1.5 Numerical Expression Evaluation

This subsection discusses the evaluation of numerical expressions. The evaluation process involves checking the semantics of the operands and determining the result based on the type of the operands. The subsection includes examples and illustrations to help understand the evaluation process.

---

7.1.12.4.1.6 Function Expression Evaluation

This subsection discusses the evaluation of function expressions. The evaluation process involves checking the semantics of the operands and determining the result based on the type of the operands. The subsection includes examples and illustrations to help understand the evaluation process.

---

7.1.12.4.1.7 Logical Expression Evaluation

This subsection discusses the evaluation of logical expressions. The evaluation process involves checking the semantics of the operands and determining the result based on the type of the operands. The subsection includes examples and illustrations to help understand the evaluation process.
Glance at Semantics: +

**operator +**

**ToNumber**

**ToString**

**ToPrimitive**
Glance at Semantics: +

Operator +

ToNumber

ToString

ToPrimitive

GetMethod

GetV

ToObject

Call
Glance at Semantics: +

- **Operator +**
- **ToNumber**
- **ToString**
- **ToPrimitive**
- **GetMethod**
- **GetV**
- **ToObject**
- **Call**
- **JS Property Access**
- **User Getter Method**
- **Proxy Intercession**
Glance at Semantics: +

12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + MultiplicativeExpression

1. Let \( lref \) be the result of evaluating AdditiveExpression.
2. Let \( lval \) be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let \( rref \) be the result of evaluating MultiplicativeExpression.
5. Let \( rval \) be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let \( lprim \) be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let \( rprim \) be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let \( lstr \) be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let \( rstr \) be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating \( lstr \) and \( rstr \).
12. Let \( inum \) be ToNumber(lprim).
13. ReturnIfAbrupt(inum).
14. Let \( rnum \) be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to \( inum \) and \( rnum \). See the Note below 12.7.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

Local outcome

- Number Conversion, Number Add
- String Conversion, String Add

Side effects

- JS property access
- User method invocations
- Proxy method invocations
Challenge: untyped variables and operations

```javascript
function add(a, b) {
    return a + b;
}
add(1, 2);
add("foo", 1);
add(1, "foo");
add({foo: ""}, 1);
add("hello", {toString: () => "me"});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values
Challenge: eval

function add(a, b) {
    return eval(a) + b;
}
add(1, 2);
add("b = 30", 1);

- The eval operator evaluates a string as if the code was injected directly into the scope
- Can modify locals, introduce new locals, and other horrible things
Challenge: eval

function add(a, b) {
    return eval(a) + b;
}
add(1, 2);
add("b = 30", 1);

- The eval operator evaluates a string as if the code was injected directly into the scope
- **Can modify locals**, introduce new locals, and other horrible things
function one(a, b) {
    var x = a + y;
    var y = 3;  // funky scoping
}

with (o) { print(x); }  // with scopes

function doit(x) {
    print(arguments);  // arguments objects
}

function* all(x) {
    for (y in x) yield y;  // generators
}
The V8 Approach
What makes V8 unique and interesting?

- V8 was the first really fast JavaScript Virtual Machine
  - Launched with Chrome in 2008
  - 10x faster than competition at release
  - 10x faster today than 2008
- Efficient object model using “hidden classes,” a technique from Self VM
- JITs galore
  - Fast AST-walking JIT compiler: fullcodegen (2008) with inline caching
  - Optimizing JIT compiler: Crankshaft (2010) with type feedback and deoptimization
  - Optimizing JIT compiler: TurboFan (2015) with type and range analysis, sea of nodes
- GCs galore
  - Evolution from simple generational collector to incremental and concurrent collector
  - Scheduling GC to reduce jank and save memory
V8 Approach: parsing

- Parsing has to be fast
  - Parsing JS is hard: hand-written, recursive descent parser

- Two modes:
  - preparse (detect structure only)
  - full (build AST) ~3x slower

- Lazy parsing:
  - A full parse of a function isn’t done until needed to execute it
  - Preparser finds boundaries of functions to quickly parse them later

- Streaming parsing:
  - Parse while script is downloading over the wire
V8 Approach: lazy compilation

JavaScript source

Abstract Syntax Tree

if
  a
  return
  0
else
  return
  2

unoptimized "fullcode" compiler

machine code
function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}
var x = new MyObject("string", 0);
x.extra = 44;
function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}
var x = new MyObject("string", 0);
x.extra = 44;
V8 Approach: object model

```javascript
function MyObject(name, data) {
  this.name = name;
  this.data = data;
  return this;
}

var x = new MyObject("string", 0);

x.extra = 44;
```
function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}
var x = new MyObject("string", 0);
var x = new MyObject("string", 0);
x.extra = 44;
V8 Approach: object model

function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}

var x = new MyObject("string", 0);
x.extra = 44;
V8 Approach: object model

```javascript
function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}
var x = new MyObject("string", 0);
x.extra = 44;
```
V8 Approach: object model

Dynamically estimated “slack tracking”

Statically estimated “expected number of properties”
V8 Approach: map forest
V8 Approach: map forest

potentially stable map
V8 Approach: object model

```javascript
function MyObject(name, data) {
    this.name = name;
    this.data = data;
    return this;
}
MyObject.prototype.print =
    function() {
        print("name: " + this.name);
        print("data: " + this.data);
    }
var bar = new MyObject("foo", 9);
```

![Diagram showing object model representation of MyObject and bar object]
V8 Approach: object model

MyObject.prototype

map
  "foo"
  9

__proto__
  8
  2
  name
  0
  data
  4

_bar_

map
  func
  func

__proto__
  8
  2
  print
  4
__maps__
V8 Approach: object model

```
map
  “foo”
  9

__proto__
  map
    func
    func
    func
    name
    8
    2
    0
    data
    4

MyObject.prototype

__proto__
  map
    func
    func
    print
    8
    2
    4
    __maps__

bar
```
V8 Approach: untyped variables and operations

function add(a, b) {
  return a + b;
}
add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add("foo", bar);

Dynamically record types of inputs to overloaded operations
V8 Approach: untyped variables and operations

function add(a, b) {
    return a + b;
}
add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add("foo", bar);

Dynamically record types of inputs to overloaded operations

Most dynamism is site-specific and stable. Normally safe to assume that what happened last time will happen the next time.
function add(a, b) {
    return a + b;
}

add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add("foo", bar);

"Usually numbers" they said!

Except they lied!
Always have a backup plan.
V8 Approach: adaptive optimization

```javascript
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```
V8 Approach: adaptive optimization

function run(a, b) {
  for (var i = 0; i < 100; i++) {
    var x = new Adder(a, b);
    x.add(i);
  }
  return x.result();
}
function run(a, b) {
  for (var i = 0; i < 100; i++) {
    var x = new Adder(a, b);
    x.add(i);
  }
  return x.result();
}
V8 Approach: adaptive optimization

```javascript
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```
V8 Approach: adaptive optimization

```javascript
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```
V8 Approach: adaptive optimization

```javascript
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```
function run(a, b) {
  for (var i = 0; i < 100; i++) {
    var x = new Adder(a, b);
    x.add(i);
  }
  return x.result();
}
A Zoo of Tiers

FullCodeGen
Unoptimized compiler

CrankShaft
optimizing compiler

TurboFan
optimizing compiler
A Zoo of Tiers (4)

TurboFan optimizing compiler

Ignition interpreter

- Faster startup!
- Saves memory!
- Still portable!

(11 supported TurboFan archs)
The Impossible Garbage Collection Triad

- High Throughput
- Low Latency
- Low Memory Overhead
V8 Garbage Collection

S  Scavenger (~0-10 ms)
I  Incremental Marking (~0.01-CONFIGURABLE ms)
M  Final Mark-Compact Collection (~4-40 ms)
F  Full Mark-Compact Collection (>40ms)

JavaScript Execution Time

Start Mark-Compact

Finish Mark-Compact
Estimating GC pauses

S  Scavenger (~0-10 ms)
I  Incremental Marking (~0.01-CONFIGURABLE ms)
M  Final Mark-Compact Collection (~4-40 ms)
F  Full Mark-Compact Collection (>40ms)

JavaScript Execution Time

Start Mark-Compact

Finish Mark-Compact
Estimating GC pauses

- **S** Scavenger (~0-10 ms)
- **I** Incremental Marking (~0.01-CONFIGURABLE ms)
- **M** Final Mark-Compact Collection (~4-40 ms)
- **F** Full Mark-Compact Collection (>40ms)

JavaScript Execution Time:

- **S** Scavenger (~0-10 ms)
- **I** Incremental Marking (~0.01-CONFIGURABLE ms)
- **M** Final Mark-Compact Collection (~4-40 ms)
- **F** Full Mark-Compact Collection (>40ms)
Latency versus Memory Overhead

- **Foreground tab**
  - Latency is critical
  - New frames are drawn every 16.66 ms when animation or scrolling happens
  - Reducing memory becomes important as soon as the tab becomes inactive

- **Background tabs**
  - Memory consumption more important than latency
  - Idle tabs can be aggressively garbage collected to save memory
Idea: Make garbage collection invisible

When is the best time to do a GC?

When nobody is looking.

Using camera to track eye movement
When subject looks away do a GC.
Life of an animation Frame

Main Thread:
- JavaScript
- Garbage Collection
- JavaScript
- Commit
- IDLE TIME

Compositor:
- Begin Frame
- Commit
- Manage Tiles
- Draw

Start Frame: 16.6 ms
End Frame: MISSED FRAME
Life of a frame

1. JS
2. IDLE
3. JS
4. IDLE
5. JS
6. IDLE
7. JS
8. GC
9. JS
10. IDLE

16ms 16ms 16ms 16ms

MISSED FRAME

11. JS
12. IDLE
13. JS
14. GC
15. IDLE
16. JS
17. IDLE
18. JS
19. IDLE
20. JS

16ms 16ms 16ms 16ms
Latency-driven Idle Time GC Scheduling (PLDI16)

- V8 heuristics tries to estimate:
  - average young generation collection speed/MB
  - average incremental marking speed/MB
  - average finalization of mark-compact speed/MB
- V8 registers an *idle garbage collection task* in the Chrome scheduler when a given garbage collection operation should happen soon
- The task scheduler will execute it when there is idle time
  - apportioning up to 50ms to perform garbage collection
Telemetry Infinite Scrolling Benchmarks

![Bar chart showing garbage collection work performed during idle time in percent.](chart.png)
WebAssembly (demo)
Motivation for WebAssembly

● Big pressure to bring native code to the web
  ○ Competition with installed mobile apps (Android, iOS)
  ○ Big-time OpenGL apps: games, CAD programs, maps
  ○ Extensibility: audio/video codecs

● Existing solutions fall short
  ○ JavaScript increasing contortions to serve as a compilation target
  ○ PNaCl encountered heavy industry resistance

● Demand for new language capabilities limited by JS bottleneck
  ○ SIMD
  ○ SharedArrayBuffer
  ○ Threads
asm.js? what’s that?

a = x + y

Normal JavaScript

x: int32
y: int32

ToNumber?
ToString?
StringAdd?
IntegerAdd?
DoubleAdd?

asm.js

a = x + y | 0

Int32Add
a: int32

asm.js

x: float64
y: float64

a = +(x + y)

Float64Add
a: float64
var buffer = new ArrayBuffer(16 * 1024 * 1024);
function module(buffer, stdlib) {
    “use asm”;
    var heap8 = new Int8Array(buffer);
    function foo(a) {
        a = a | 0;
        return heap8[a] + 1 | 0;
    }
    return {foo: foo}
}

var mod = module(buffer, {print: print});
mod.foo(100);
asm.js? what’s that? (3)

- Emscripten: A POSIX-like platform with
  - Toolchain based on forked LLVM
  - libc
  - OpenGL (on top of WebGL)
  - a community
  - Game engines
  - Applications
  - Benchmarks
asm.js? what’s that? (4)

- 2 engines specially recognize asm.js subset and validate that subset
  - Mozilla Firefox - pioneer
  - Microsoft Edge - fast follow
- V8 uses TurboFan’s advanced type analysis to recover the same information
  - Within ~X% of custom solution on most benchmarks
  - No inter-procedural optimizations
  - Crossover with optimizing normal JavaScript
- V8 can validate asm.js subset and internally translate to WebAssembly
What is WebAssembly?

- A compilation target for native
  - C/C++, other languages -> WASM
- A new capability for the web
  - More than just compressed asm.js
  - float32, int64, threads*, SIMD*
- A complement to JavaScript
  - interface to/from JS code
  - integrate with WebAPIs
- Performance guarantee (ish)
  - Fast calling conventions
  - no boxing, no GC
  - AOT

What is WebAssembly not?

- A value judgment about languages
  - JavaScript vs C++ vs Java vs Dart
- The backend of some C compiler
  - LLVM bitcode, gcc GIMPLE, sea of nodes
- A programming language
  - generated and manipulated by tools
- A separate VM within Chrome
  - instead: built on TurboFan and V8
V8 Pipeline Design (asm.js)

JavaScript source → JSFunction → fullcode → unoptimized code → TurboFan → hot asm.js → optimized code
V8 Pipeline Design + WASM

JavaScript source → fullcode → TurboFan → backend → optimized code

- JavaScript analysis + lowering
- hot asm.js
- decode
- wasm binary
V8 Pipeline Design + asm.js + WASM

JavaScript source → asm.js module → verify + encode → wasm binary → optimized code

TurboFan: javascript analysis + lowering → backend
WebAssembly in a nutshell

- **Data Types**
  - void i32 i64 f32 f64

- **Functions**
  - Flat, single global table
  - Static binding
  - Indirect calls through table

- **State: linear memory**
  - large, bounds-checked array

- **Trusted execution stack**

- **Data Operations**
  - i32: + - * / % << >> >>> etc
  - i64: + - * / % << >> >>> etc
  - f32: + - * / sqrt ceil floor
  - f64: + - * / sqrt ceil floor
  - conversions
  - load store
  - call_direct call_indirect

- **Structured Control Flow**
  - if loop block br switch
WebAssembly trusted and untrusted state

- Linear memory
- Functions
- Indirect function table
- Execution stack
● C compiler translates pointers to i32 indices
● C compiler places addressable stack in memory
● asm.js bounds checks (~5% overhead)
WebAssembly binary code

● Goals:
  ○ compact => smaller than minified JS
  ○ easy to verify => one linear pass
  ○ easy to compile => one linear pass to construct IR or baseline JIT
  ○ extensible => anticipate new bytecodes and types

● Design:
  ○ AST-based post-order encoding of function bodies
  ○ All AST nodes are expressions
  ○ Optional application-specifed opcode table
Module structure

- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data
Module structure

- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

min_size = 16mb
max_size = 1gb
exported_to_js = false
Module structure

- Memory declaration
- Function signatures
  - (i32, i32) -> i32
  - (i64, i32) -> i32
  - (f32) -> i32
- Functions
- Indirect Function Table
- Initialized data
Module structure

- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

myfunc:
  <sig>
  <flags>
  <code>
Module structure

- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
  - 0: myfunc1
  - 1: myfunc2
  - 2: myfunc2
- Initialized data
Module structure

- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

0x0109de8: <data>
0x0f0a9c12: <data>
0x00034a00: <data>
Bytecode => TurboFan

- One Linear pass to construct sea of nodes
  - SSA environment tracks control and effect dependencies
  - Stack of if, blocks, and loops
  - Conservative phi insertion at loop headers
  - Reduction steps generate nodes in the IR graph

- Machine-level graph
  - Immediately suitable for code generation
  - Correct sea-of-nodes can go through scheduling
  - Can apply machine-level and machine-independent optimizations

- Fast calling convention
  - No boxing of double arguments
  - All arguments in registers
  - No extra JSFunction / context arguments
Pre-order encoding of an AST

if (a) return 0; else return 2;

return a?0:2
Decoding preorder to IR

if (a) return 0; else return 2;
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

if (a) return 0; else return 2;
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

unfinished | finished

Production stack
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack
Decoding preorder to IR

if (a) return 0; else return 2;
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

unfinished finished

reduce
Decoding preorder to IR

```cpp
if (a) return 0; else return 2;
```

Production stack

```
if
local
#0
ret
iconst
#0
ret
iconst
#2
```
if (a) return 0; else return 2;
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

unfinished finished

if
local
#0
ret
iconst
#0
ret
iconst
#2
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

unfinished | finished

shift

if | ret | const#0

if | local | #0

ret

iconst | #0

ret

iconst | #2
Decoding preorder to IR

```
if (a) return 0; else return 2;
```

Production stack

```
if
local
#0
ret
iconst
#0
ret
iconst
#2
```
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

```plaintext
if
local #0
ret
iconst #0
ret
iconst #2
```

unfinished

finished

reduce
Decoding preorder to IR

```c
if (a) return 0; else return 2;
```
Decoding preorder to IR

```java
if (a) return 0; else return 2;
```

Production stack:
```
if
local #0
ret
iconst #0
ret
iconst #2
```

Unfinished: 0

Finished: 1

Production stack:
```
if
local #0
ret
iconst #0
ret
iconst #2
```
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack
Decoding preorder to IR

```c
if (a) return 0; else return 2;
```

Production stack

```
if
local #0
ret
iconst #0
ret
iconst #2
```
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

if
local
#0
ret
iconst
#0
ret
iconst
#2

unfinished
finished

advance
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack:

unfinished
finished

shift

if    ret    const#2
0
1
2
3
4
5
6
7
8

if
local
#0
ret
iconst
#0
ret
iconst
#2
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

if

if
data
local
#0
ret
iconst
#0
ret
iconst
#2

unfinished
finished

reduce
Decoding preorder to IR

\[
\text{if (a) return 0; else return 2;}
\]
Decoding preorder to IR

```java
if (a) return 0; else return 2;
```

Production stack:

```
if
local
#0
ret
iconst
#0
ret
iconst
#2
```

Diagram:

```
if
  a
    return
      0
else
    return
      2
```
Decoding preorder to IR

if (a) return 0; else return 2;
Decoding preorder to IR

if (a) return 0; else return 2;

Production stack

<table>
<thead>
<tr>
<th>unfinished</th>
<th>finished</th>
</tr>
</thead>
</table>

reduce
Decoding preorder to IR

```
if (a) return 0; else return 2;
```
Decoding preorder to IR

```c
if (a) return 0; else return 2;
```

Production stack:
```
0: if
1: local
2: #0
3: ret
4: iconst
5: #0
6: ret
7: iconst
8: #2
```

Unfinished: ♻️ Finished: 🍁
Bytecode ⇒ TurboFan

- One Linear pass to construct sea of nodes
  - SSA environment tracks control and effect dependencies
  - Stack of if, blocks, and loops
  - Conservative phi insertion at loop headers
  - Reduction steps generate nodes in the IR graph

- Machine-level graph
  - Immediately suitable for code generation
  - Correct sea-of-nodes can go through scheduling
  - Can apply machine-level and machine-independent optimizations
function (x) {
  return x ? 1 : 2;
}
TurboFan SSA Environment

TurboFan Nodes

state  control  effect  locals
Using the SSA environment

bytecode: local[0] = local[0] + local[1]
Minimal SSA Renaming in one pass

```c
if (a) return 0; else return 2;
```

Virtual CFG
if (a) return 0; else return 2;
Minimal SSA Renaming in one pass

```
if (a) return 0; else return 2;
```

Virtual CFG

```
begin
if
   a
   return 0
else
   return 2
end
```
Stack of SSA environments
The same great AST: now in postorder!

Function Bodies
Post-order encoding of an AST

return 3 + x * 4
Decoding post-order to an AST

\[
\text{return } 3 + x \times 4
\]
return 3 + x * 4
Decoding post-order to an AST

return 3 + x * 4

const #3
advance

const #3

finished
Decoding post-order to an AST

return 3 + x * 4
Decoding post-order to an AST

\[ \text{return } 3 + x \times 4 \]
Decoding post-order to an AST

```
return 3 + x * 4
```

![Diagram of an AST for the expression](image)

- `return`: function declaration
- `3`: constant
- `x`: variable
- `+`: binary operator
- `*`: binary operator
- `local`: local variable declaration
Decoding post-order to an AST

\[ \text{return } 3 + x \times 4 \]
Decoding post-order to an AST

```
return 3 + x * 4
```

```
0 1 2 3 4 5 6 7 8
iconst #3 iconst #4 local #0 imul iadd ret
```

return 3 + x * 4
Decoding post-order to an AST

```
return 3 + x * 4
```

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```

```
const#4 local
const#3 imul
push finished
```
Decoding post-order to an AST

```
return 3 + x * 4
```

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```

```
const#4 local
```

```
const#3 imul
```

```
return 3 + x * 4
```

```
0 1 2 3 4 5 6 7 8
iconst #3 #4 local #0 imul iadd ret
```

```
const#4
```
Decoding post-order to an AST

```
return 3 + x * 4
```

```
+  
  
  4
  
  3
  
  x
  
  *  
  
  ret
  
iadd
  
imul
  
#0
  
local
  
#4
  
iconst
  
const#4
  
const#3
  
```

finished

```
0  
1  
2  
3  
4  
5  
6  
7  
8
```

pop
Decoding post-order to an AST

return 3 + x * 4

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```

```
finished
const#4 local
const#3 imul
iadd
push
```
Decoding post-order to an AST

```
return 3 + x * 4
```

```
{advance}

0: iconst #3
1: iconst #4
2: local #0
3: imul
4: iadd
5: ret

return

const#4
local
const#3
imul
iadd

finished
```
Decoding post-order to an AST

```
return 3 + x * 4
```

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```
Decoding post-order to an AST

```
return 3 + x * 4
```

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```

```
const#4 local
const#3 imul
iadd
ret
```

```
finished
```

```
push
```
Decoding post-order to an AST

return 3 + x * 4

```
iconst #3
iconst #4
local #0
imul
iadd
ret
```

```
const#4
local
imul
iadd
ret
```

finished

```
return 3 + x * 4

3

4

x

*

+

return

finish
```
Decode+Verify performance

return 3 + x * 4

Preorder vs Postorder (arithmetic)
Decode+Verify performance

```python
select(2, 3, x)
```

![Graph showing Preorder vs Postorder for select operation]
Decode+Verify performance

block(block(br_if $0 x) br($1, #3)) #2)
Postorder encodings of control

block
br
br_if
if
if_else
tableswitch
Preorder vs. Postorder block

(block x, y, z)
Preorder vs. Postorder block verification

(block x, br $0, z)

preorder

bracketed

single-pass verification
Preorder vs. Postorder if/else

(if_else x, y, z)

preorder

in-order
Preorder vs. Postorder if/else

(if_else x, y, z)

preorder

in-order

single-pass verification
Preorder vs. Postorder if/else

(if x, y)

✔

preorder

✔
in-order

single-pass verification
Preorder vs. Postorder tables\texttt{switch}

\[(\text{tables\texttt{switch}} \ x, \ y, \ z, \ w)\]
Preorder vs. Postorder tables

\[(\text{tableswitch } x, y, z, w)\]
**WebAssembly binary code**

- **Goals:**
  - compact => smaller than minified JS
  - easy to verify => one linear pass
  - easy to compile => one linear pass to construct IR or baseline JIT
  - extensible => anticipate new bytecodes and types

- **Did we deliver?**
  - Fast single-pass decode+verify (> 100MB/s)
  - Single-pass to compiler IR demonstrated (V8/TurboFan)
  - Fast optimizing compiler (1.8MB/s single thread, 7MB/s with 8 threads)
  - Within 20% of native code execution speed (geomean; vs 80% for asm.js)
  - Single-pass compiler in development (Mozilla)
Compiling WASM vs. Compiling asm.js

- JavaScript is not statically typed
  - Values have types, not variables
  - 8 is a number, “foo” is a string
  - All basic operators (+ - / * % << >>) are overloaded or have implicit conversions

- All arithmetic is done in 64-bit floating point
  - Empirically most programs use small integers (<= 31 bits)
  - Overflow to double causes bailout to slow path, allocation, etc
  - Troublesome cases {-0.0 NaN Infinity -Infinity}

- Type “annotations” in asm.js
  - a + b | 0 is integer arithmetic
  - +(a + b) is double arithmetic
  - (a >>> 0) < (b >>> 0) is an unsigned comparison
Type and Range Analysis (asm.js)

- **Typing alone**:
  - \( x: \text{Int} \)
  - \( y: \text{Int} \)
  - \( +: \text{Num} \)

- **Typing + range analysis**:
  - \( x: \text{Int in [0, 9]} \)
  - \( y: \text{Int in [11, 15]} \)
  - \( +: \text{Int in [11, 26]} \)

- **Typing alone**:
  - \( x: \text{Int} \)
  - \( y: \text{Int} \)
  - \( \phi: \text{Int} \)

- **Typing + range analysis**:
  - \( x: \text{Int in [0, 9]} \)
  - \( y: \text{Int in [11, 15]} \)
  - \( \phi: \text{Int in [0, 15]} \)
Typed lowering as Reduction (asm.js)

\[
\begin{align*}
\text{NumAdd} & : \text{Num} + \text{Num} \\
\text{StringAdd} & : \text{String} + \text{Num} \\
\text{Int32Add} & : \text{Int} + \text{Int} \\
\text{BinopIC} & : \text{Any} + \text{Any}
\end{align*}
\]
WASM = no lowering necessary!
General Reductions

3 + 5

x 0 +

x 4 *

x \phi

3 + 5

constant folding

strength reduction

strength reduction

phi simplification

algebraic reassociation

8

x

x 2

x <<

x 12

x +
General Reductions (2)
WebAssembly Status

- LLVM backend upstream
- Lots of tools
- Reference implementation (spec) in Standard ML
- 3 Browser engines have native support in various stages
  - Google Chrome Beta: fully spec compliant on all architectures, behind a flag
  - Mozilla Firefox: optimized for ia32 and x64, behind a flag
  - Microsoft Edge: support in an experimental build
- MVP (Version 1.0) expected to be shipped this summer
- Standardization expected by the end of the year

https://github.com/WebAssembly/
Questions?