Ignition: Jump-starting an Interpreter for V8

Ross McIlroy
Google London
Agenda

- Why we all love JavaScript
- The V8 approach
- How to retrofit an interpreter into a moving engine
Why we all love JavaScript...
JavaScript

- The language of the Web
JavaScript

- The language of the Web
- Programs are distributed as source - parsing and compiling must be fast
JavaScript

- The language of the Web
- Programs are distributed as source - parsing and compiling must be fast
- Untyped: variables and properties do not have types, values do
JavaScript

- The language of the Web
- Programs are distributed as source - parsing and compiling must be fast
- Untyped: variables and properties do not have types, values do
- Prototype-based object model
JavaScript

- The language of the Web
- Programs are distributed as source - parsing and compiling must be fast
- Untyped: variables and properties do not have types, values do
- Prototype-based object model
- Functional features with closures
JavaScript

- The language of the Web
- Programs are distributed as source - parsing and compiling must be fast
- Untyped: variables and properties do not have types, values do
- Prototype-based object model
- Functional features with closures
- A smattering of interesting features
  - `eval()` allows dynamic execution of runtime generated statements within a function
  - Weird scoping rules
  - Default values and implicit type coercion
  - ...
Something Simple

```javascript
function add(a, b) {
    return a + b;
}
```
Something Simple

```javascript
function add(a, b) {
    return a + b;
}

add(1, 2); // 3
```

Integer addition
Something Simple

```javascript
function add(a, b) {
    return a + b;
}
add(1, 2);           // 3
add(1.2, 3.14);      // 4.34
```

**Integer addition**

**Floating point addition**
function add(a, b) {
    return a + b;
}

add(1, 2);       // 3
add(1.2, 3.14);  // 4.34
add("hello", "world"); // "helloworld"
function add(a, b) {
    return a + b;
}
add(1, 2);     // 3
add(1.2, 3.14); // 4.34
add("hello", "world"); // "helloworld"
add(1, true);   // 2
Something Simple

```javascript
function add(a, b) {
    return a + b;
}
add(1, 2);       // 3
add(1.2, 3.14);  // 4.34
add("hello", "world");  // "helloworld"
add(1, true);    // 2
add("foo", true); // "footrue"
```
Something Simple

```javascript
function add(a, b) {
    return a + b;
}

add(1, 2);       // 3
add(1.2, 3.14);  // 4.34
add("hello", "world"); // "helloworld"
add(1, true);    // 2
add("foo", true); // "footrue"
var bar = {toString:() => "bar"};
add("foo", bar); // "foobar"
```

- Integer addition
- Floating point addition
- String addition
- Type coercion
- `toString()` / `valueOf()`
A Glance at Semantics

12.7.3.1 Runtime Semantics: Evaluation

\[
\text{AdditiveExpression} : \text{AdditiveExpression} + \text{AdditiveExpression}
\]

1. Let \( lRef \) be the result of evaluating \( \text{AdditiveExpression} \).
2. Let \( lVal \) be \( \text{GetValue}(lRef) \).
3. ReturnIfAbrupt(\( lVal \)).
4. Let \( rRef \) be the result of evaluating \( \text{MultiplicativeExpression} \).
5. Let \( rVal \) be \( \text{GetValue}(rRef) \).
6. ReturnIfAbrupt(\( rVal \)).
7. Let \( lPrim \) be \( \text{ToPrimitive}(lVal) \).
8. ReturnIfAbrupt(\( lPrim \)).
9. Let \( rPrim \) be \( \text{ToPrimitive}(rVal) \).
10. ReturnIfAbrupt(\( rPrim \)).
11. If \( \text{Type}(lPrim) \) is String or \( \text{Type}(rPrim) \) is String, then
    a. Let \( lStr \) be \( \text{ToString}(lPrim) \).
    b. ReturnIfAbrupt(\( lStr \)).
    c. Let \( rStr \) be \( \text{ToString}(rPrim) \).
    d. ReturnIfAbrupt(\( rStr \)).
    e. Return the String that is the result of concatenating \( lStr \) and \( rStr \).
12. Let \( lNum \) be \( \text{ToNumber}(lPrim) \).
13. ReturnIfAbrupt(\( lNum \)).
14. Let \( rNum \) be \( \text{ToNumber}(rPrim) \).
15. ReturnIfAbrupt(\( rNum \)).
16. Return the result of applying the addition operation to \( lNum \) and \( rNum \). See the Note below 12.7.5.

**NOTE 1** No hint is provided in the calls to \( \text{ToPrimitive} \) 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.
A Glance at Semantics

12.7.1.1 Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + AdditiveExpression
1. Let left be the result of evaluating AdditiveExpression.
2. Let right be the result of evaluating AdditiveExpression.
3. Return If left is a Number, or right is a Number, then let result be left + right, otherwise throw a TypeError.

ToPrimitive

Table 9 — ToPrimitive Conversion

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Return null</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return the symbol object</td>
</tr>
<tr>
<td>Object</td>
<td>Return the value following the slot.</td>
</tr>
<tr>
<td>anything (other)</td>
<td>Throw a TypeError exception</td>
</tr>
</tbody>
</table>

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

1. If ToPrimitive was not passed, let O be "default".
2. If O is a Symbol or a String, let O be "String".
3. If O is a Number, let O be "number".
4. Let result be O.
5. If ToPrimitive is no widead, then
   a. Let result be CallMethod(O, input, others).
   b. Return ToMethod(result).
   c. If ToMethod is an Object, return result.
   d. If ToMethod is an Array, return result.
   e. If ToMethod is a Function, return result.
   f. Throw a TypeError exception.

NOTE
When ToPrimitive is called with no last, it is generally behaves as if the last were Number. However, objects may override this behavior by defining a ToPrimitive method. If the objects defined this operation only for these objects (see 12.1.2.1) and Symbol objects (see 10.4.2.2b) override the default ToPrimitive behavior. Date objects are not one of the last being String.
A Glance at Semantics

operator +

ToString

ToPrimitive

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 ToNumber was given. Exception objects handle the absence of a hint as if the hint String was given. Exception objects should handle a hint as if the hint String was given.

NOTE 2: See: "Differences from step 8 of the Abstract Relational Comparison algorithm (7.2.1.1)" for the logical operation instead of the logical and operation.

Google
A Glance at Semantics

operator +

ToNumber

ToPrimitive

ToString

NOTE 1: No hint is provided in the calls to ToPrimitive in rows 7 and 8. All standard objects except Date objects handle the absence of a hint in the hint column in such cases. Date objects may handle the absence of a hint in some other manner.

NOTE 2: See 11.1.12, section 11.1.3.1 of the Abstract Relational Comparison algorithm (7.2.11), for using the logical-or operator instead of the logical-and operator.

Source: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis non erat sem.
A Glance at Semantics
A Glance at Semantics

operator +

ToNumber

ToString

ToPrimitive

GetMethod

GetV

ToObject

Call

Arbitrary Javascript
A Glance at Semantics

operator +

ToNumber

ToString

ToPrimitive

GetMethod

GetV

ToObject

Call

Arbitrary Javascript
Everything’s a Function

```javascript
function Person(name) {
    this.name = name;
}
```

An object’s constructor is just a function
Everything’s a Function

```javascript
function Person(name) {
    this.name = name;
}
Person.prototype.toString = function() { return this.name; }
```

Method’s are installed on the `prototype` of an object.
Everything’s a Function

```javascript
function Person(name) {
    this.name = name;
}
Person.prototype.toString = function() { return this.name; }
var jill = new Person("Jill");
print(jill); // "Jill"
```

Objects are instantiated by “new <Function>(...)”
Everything’s a Function

```javascript
function Person(name) {
    this.name = name;
}
Person.prototype.toString = function() { return this.name; }

function Student(name, grade) {
    Person.call(this, name);
    this.grade = grade;
}
Student.prototype.__proto__ = Person.prototype;

var tom = new Student("Tom", 72);
print(tom); // "Tom"
```

Inheritance emulated by prototype chaining
Everything’s a Function

```javascript
function Person(name) {
    this.name = name;
}
Person.prototype.toString = function() { return this.name; }

function Student(name, grade) {
    Person.call(this, name);
    this.grade = grade;
}
Student.prototype.__proto__ = Person.prototype;
var tom = new Student("Tom", 72);

// This line is commented out in the slide, but if uncommented, it would change tom's prototype.
tom.__proto__ = Object.prototype;
print(tom); // “[object Object]”
```

Which is completely dynamic....
Except when it’s a Closure

```javascript
function Counter(start) {
    var count = 0;
    return {
        next: function() { return start + count++; }
    }
}
```
Except when it’s a Closure

```javascript
function Counter(start) {
  var count = 0;
  return {
    next: function() { return start + count++; }
  }
}

var counter = Counter(5);
print(counter.next() + " -> " + counter.next());  // 5 -> 6
```

Closures over parameters, and mutable local variables
Fun with `eval()`

```javascript
function func(a, b) {
  return eval(a) + (b == 0 ? 0 : func(a, --b));
}

func("1", 3);  // 4
```

Executes string within the context of the calling function
Fun with eval()

```javascript
function func(a, b) {
  return eval(a) + (b == 0 ? 0 : func(a, --b));
}

func("1", 3); // 4
func("b = 0", 200); // 0
```

- Executes string within the context of the calling function
- Can modify locals or introduce new ones
Fun with eval()

```javascript
function func(a, b) {
    return eval(a) + (b == 0 ? 0 : func(a, --b));
}
func("1", 3); // 4
func("b = 0", 200); // 0
func("func = function() {
    return 'bar'
}; 'foo'", 50); // "foobar"
```

- Executes string within the context of the calling function
- Can modify locals or introduce new ones
- Or do crazy things...
The V8 Approach
V8 History

● V8 was the first really fast JavaScript Virtual Machine
  ○ Launched with Chrome in 2008
  ○ 10x faster than competition at release
  ○ 10x faster today than in 2008

● 2008 - Full-Codegen
  ○ Fast AST-walking JIT compiler with inline caching

● 2010 - Crankshaft
  ○ Optimizing JIT compiler with type feedback and deoptimization

● 2015 - TurboFan
  ○ Optimizing JIT compiler with type and range analysis, sea of nodes
Compiler Pipeline (2008)
Full-Codegen in a nutshell

```javascript
function Sum(point) {
    return point.x + point.y;
};
```
Full-Codegen in a nutshell

```javascript
function Sum(point) {
  return point.x + point.y;
}
```
Full-Codegen in a nutshell

```
return
+
Load Property
  point  "x"
Load Property
  point  "y"
```
Full-Codegen in a nutshell

... ; prologue
mov eax, [ebp + 0x10] ; point
Full-Codegen in a nutshell

...                      ; prologue
mov eax, [ebp + 0x10]    ; point
mov ecx, 0x56a79431      ; “x”

Full-Codegen
Full-Codegen in a nutshell

... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; “x”
call $LoadNamedProperty
push eax

Full-Codegen
Full-Codegen in a nutshell

... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; “x”
call $LoadNamedProperty
push eax

mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; “y”
call $LoadNamedProperty

return

+  

Load Property  

| point | “x” |

Load Property  

| point | “y” |

Full-Codegen
Full-Codegen in a nutshell

```
... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; “x”
call $LoadNamedProperty
push eax
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; “y”
call $LoadNamedProperty
pop edx
call $BinaryOpAdd
...
```

Full-Codegen
Full-Codegen in a nutshell

... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; "x"
call $LoadNamedProperty
push eax
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; "y"
call $LoadNamedProperty
pop edx
call $BinaryOpAdd
...
Hidden Classes

```javascript
function Point(x, y) {
    this.x = x;
    this.y = y;
}
```

Hidden classes was a technique from Self VM.
function Point(x, y) {
    this.x = x;
    this.y = y;
};

var point = new Point(3, 5);
Hidden Classes

```javascript
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var point = new Point(3, 5);
```
function Point(x, y) {
    this.x = x;
    this.y = y;
}

var point = new Point(3, 5);
function Point(x, y) {
    this.x = x;
    this.y = y;
};

var point = new Point(3, 5);
Inline Caches (ICs)

... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; "x"
call $LoadNamedProperty
push eax
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; "y"
call $LoadNamedProperty
pop edx
call $BinaryOpAdd

UNINITIALIZED_LOAD_IC
- Call into runtime
- Determine object layout
- Load property with <name>
- Generate specialized IC
- Back-patch original call
Inline Caches (ICs)

```
... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; "x"
call $LoadNamedProperty
push eax
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; "y"
call $LoadNamedProperty
pop edx
call $BinaryOpAdd
...
Inline Caches (ICs)

... ; prologue

mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; “x"
call $LoadNamedProperty
push eax

mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; “y"
call $LoadNamedProperty

call $BinaryOpAdd

...
Inline Caches (ICs)

... ; prologue
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a79431 ; “x”
call $LoadNamedProperty
push eax
mov eax, [ebp + 0x10] ; point
mov ecx, 0x56a71251 ; “y”
call $LoadNamedProperty
pop edx
call $BinaryOpAdd
...

MONOMORPHIC_LOAD_IC_X

... ; Check object’s map is...
... ; Point type, or bailout
mov eax, [eax + 0x4]
ret

MONOMORPHIC_LOAD_IC_Y

... ; Check object’s map is...
... ; Point type, or bailout
mov eax, [eax + 0x8]
ret

BINARY_OP_ADD_IC
Compiler Pipeline (2010)

Source

Parser → Full-codegen → Unoptimized Code → Baseline

Parser → Crankshaft → Optimized Code → Optimized
function Sum(point) {
    return point.x + point.y;
};
function Sum(point) {
    return point.x + point.y;
};

Sum(new Point(1, 2));
Sum(new Point(100, 6));
Sum(new Point(0.5, 30));
Sum(new Point(0.5, 30));
A Little on Crankshaft

```javascript
function Sum(point) {
  return point.x + point.y;
}

Sum(new Point(1, 2));
Sum(new Point(100, 6));
Sum(new Point(0.5, 30));
Sum(new Point(0.5, 30));
```

Always a number
function Sum(point) {
    return point.x + point.y;
}

Sum(new Point(1, 2));
Sum(new Point(100, 6));
Sum(new Point(0.5, 30));
Sum(new Point(0.5, 30));
A Little on Crankshaft

```javascript
function Sum(point) {
    return point.x + point.y;
}

Sum(new Point(1, 2));
Sum(new Point(100, 6));
Sum(new Point(0.5, 30));
Sum(new Point(0.5, 30));
```

Always a number

 Inline property load

 Elide map checks

 Always Point

 Inline FP addition

Type analysis

Escape analysis

GVN

Inlining
A Little on Crankshaft

```javascript
function Sum(point) {
    return point.x + point.y;
}

Sum(new Point(1, 2));
Sum(new Point(100, 6));
Sum(new Point(0.5, 30));
Sum(new Point(0.5, 30));
Sum(new StringPair("foo", "bar");
```
Deoptimization - Always Have a Backup Plan

- Deopt points inserted before speculative optimizations

- Crankshaft needs to model Full-Codegen’s execution to rebuild a stack frame for the deopt point
Compiler Pipeline (2015)
Another Optimizing Compiler?

Crankshaft served us well, but has various shortcomings:

● Doesn’t scale to full modern JavaScript
  ○ try-catch, for-of, generators, async/await
Another Optimizing Compiler?

Crankshaft served us well, but has various shortcomings:

- Doesn’t scale to full modern JavaScript
  - try-catch, for-of, generators, async/await
- Relies heavily on deoptimization
  - Performance cliffs and deoptimization loops
Another Optimizing Compiler?

Crankshaft served us well, but has various shortcomings:

- Doesn’t scale to full modern JavaScript
  - try-catch, for-of, generators, async/await
- Relies heavily on deoptimization
  - Performance cliffs and deoptimization loops
- Limited static type analysis / propagation
  - Not amenable to asm.js style optimization
Another Optimizing Compiler?

Crankshaft served us well, but has various shortcomings:

- Doesn’t scale to full modern JavaScript
  - try-catch, for-of, generators, async/await
- Relies heavily on deoptimization
  - Performance cliffs and deoptimization loops
- Limited static type analysis / propagation
  - Not amenable to asm.js style optimization
- Tight coupling Full-codegen
Another Optimizing Compiler?

Crankshaft served us well, but has various shortcomings:

- Doesn’t scale to full modern JavaScript
  - try-catch, for-of, generators, async/await
- Relies heavily on deoptimization
  - Performance cliffs and deoptimization loops
- Limited static type analysis / propagation
  - Not amenable to asm.js style optimization
- Tight coupling Full-codegen
- High porting overhead
TurboFan

● Sea of Nodes
  ○ Relax evaluation order for most operations (value edges)
  ○ Skeleton of a CFG remains (control edges) and stateful operations (effect edges)
  ○ Provides better redundant code elimination and more code motion
TurboFan

- **Sea of Nodes**
  - Relax evaluation order for most operations (value edges)
  - Skeleton of a CFG remains (control edges) and stateful operations (effect edges)
  - Provides better redundant code elimination and more code motion

- **Three Level IR**
  - JavaScript: JavaScript’s overloaded operators
  - Simplified: VM operations, e.g. allocation or number arithmetic
  - Machine: Machine-level operations, e.g. `int32` addition
TurboFan

● **Sea of Nodes**
  ○ Relax evaluation order for most operations (value edges)
  ○ Skeleton of a CFG remains (control edges) and stateful operations (effect edges)
  ○ Provides better redundant code elimination and more code motion

● **Three Level IR**
  ○ JavaScript: JavaScript’s overloaded operators
  ○ Simplified: VM operations, e.g. allocation or number arithmetic
  ○ Machine: Machine-level operations, e.g. int32 addition

● **Lowering JS graph to simplified graph based on types**
  ○ Take into account *static* type information and type feedback
function (x) {
    return x ? 1 : 2;
}
Retrofitting an Interpreter into a Moving Engine
Why Interpret?

- Reduce memory usage
Why Interpret?

- Reduce memory usage
- Reduce startup time

33% of time spent parsing + compiling
Why Interpret?

- Reduce memory usage
- Reduce startup time
- Reduce complexity
Ignition - Goals

● Reduce memory usage
  ○ Compile to bytecode which is 4x smaller than machine code
  ○ Reduce overall code memory by 2x

● Reduce startup time

● Reduce complexity
Ignition - Goals

● **Reduce memory usage**
  ○ Compile to bytecode which is 4x smaller than machine code
  ○ Reduce overall code memory by 2x

● **Reduce startup time**
  ○ Faster compiling to bytecode
  ○ Reduce re-parsing for lazy compile and optimize re-compile

● **Reduce complexity**
Ignition - Goals

● Reduce memory usage
  ○ Compile to bytecode which is 4x smaller than machine code
  ○ Reduce overall code memory by 2x

● Reduce startup time
  ○ Faster compiling to bytecode
  ○ Reduce re-parsing for lazy compile and optimize re-compile

● Reduce complexity
  ○ Bytecode as source of truth
  ○ Simplify compilation pipeline
Ignition - Challenges

● Don’t regress performance
Ignition - Challenges

- Don’t regress performance
- Support 100% of the JavaScript language on 9 CPU architectures
Ignition - Challenges

- Don’t regress performance
- Support 100% of the JavaScript language on 9 CPU architectures
- Integrate with V8’s runtime (type feedback, object model, GC, etc)
Ignition - Challenges

● Don’t regress performance
● Support 100% of the JavaScript language on 9 CPU architectures
● Integrate with V8’s runtime (type feedback, object model, GC, etc)
● Support the debugger / liveedit
Ignition - Challenges

● Don’t regress performance
● Support 100% of the JavaScript language on 9 CPU architectures
● Integrate with V8’s runtime (type feedback, object model, GC, etc)
● Support the debugger / liveedit
● Support two pipelines (Crankshaft and TurboFan)
Compiler Pipeline (2015)

- Full-codegen
- Unoptimized Code
- Crankshaft
- TurboFan
- Optimized Code

Baseline

Optimized
Compiler Pipeline (2015)
Compiler Pipeline (2016)
Compiler Pipeline (2016)

- Full-codegen
- Unoptimized Code
- Ignition
- Bytecode
- TurboFan
- Optimize
- Crankshaft
- Optimized Code
Compiler Pipeline (early 2017 ?)
Compiler Pipeline (early 2017 ?)

- Ignition
- Bytecode
- Interpreted
- Full-codegen
- Optimize
- Baseline
- Unoptimized Code
- TurboFan
- Optimized
- Crankshaft
- Optimized Code
Compiler Pipeline (2017 ?)
Ignition Design Decisions

- Focus on reducing code size
  - Indirect threaded bytecode dispatch
  - Accumulator as implicit input / output
Ignition Design Decisions

● Focus on reducing code size
  ○ Indirect threaded bytecode dispatch
  ○ Accumulator as implicit input / output

● But still as fast as possible
  ○ Hand coded using (architecture-independent) macro-assembly
  ○ Register machine
Ignition Design Decisions

● Focus on reducing code size
  ○ Indirect threaded bytecode dispatch
  ○ Accumulator as implicit input / output

● But still as fast as possible
  ○ Hand coded using (architecture-independent) macro-assembly
  ○ Register machine

● Bytecode can be used to build TurboFan graphs directly
  ○ Bytecode is single source of truth
  ○ Simpler deoptimization execution modeling
function f(a, b, c) {
  var local = c - 100;
  return a + local * b;
}
Ignition Bytecode

```javascript
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}
```

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
Ignition Bytecode

```javascript
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}
```

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>[a]</td>
<td>5</td>
</tr>
<tr>
<td>a1</td>
<td>[b]</td>
<td>2</td>
</tr>
<tr>
<td>a2</td>
<td>[c]</td>
<td>150</td>
</tr>
<tr>
<td>r0</td>
<td>[local]</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Ignition Bytecode

```javascript
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}
```

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>[a]</td>
<td>5</td>
</tr>
<tr>
<td>a1</td>
<td>[b]</td>
<td>2</td>
</tr>
<tr>
<td>a2</td>
<td>[c]</td>
<td>150</td>
</tr>
<tr>
<td>r0</td>
<td>[local]</td>
<td>undefined</td>
</tr>
<tr>
<td>accumulator</td>
<td></td>
<td>undefined</td>
</tr>
</tbody>
</table>
function f(a, b, c) {
  var local = c - 100;
  return a + local * b;
}
function f(a, b, c) {
  var local = c - 100;
  return a + local * b;
}

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

| a0 [a] | 5  |
| a1 [b] | 2  |
| a2 [c] | 150|
| r0 [local] | undefined |
| accumulator | 50 |
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}

Ignition Bytecode

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>[a]</td>
<td>5</td>
</tr>
<tr>
<td>a1</td>
<td>[b]</td>
<td>2</td>
</tr>
<tr>
<td>a2</td>
<td>[c]</td>
<td>150</td>
</tr>
<tr>
<td>r0</td>
<td>[local]</td>
<td>50</td>
</tr>
<tr>
<td>accumulator</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
Ignition Bytecode

```javascript
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>a1</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a2</td>
<td>c</td>
<td>150</td>
</tr>
<tr>
<td>r0</td>
<td>local</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>accumulator</td>
<td>2</td>
</tr>
</tbody>
</table>
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}

Ignition Bytecode

<table>
<thead>
<tr>
<th></th>
<th>LdaSmi #100</th>
<th>Sub a2</th>
<th>Star r0</th>
<th>Ldar a1</th>
<th>Mul r0</th>
<th>Add a0</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>acc</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
function f(a, b, c) {
  var local = c - 100;
  return a + local * b;
}

LdaSm#100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>[a]</td>
<td>5</td>
</tr>
<tr>
<td>a1</td>
<td>[b]</td>
<td>2</td>
</tr>
<tr>
<td>a2</td>
<td>[c]</td>
<td>150</td>
</tr>
<tr>
<td>r0</td>
<td>[local]</td>
<td>50</td>
</tr>
<tr>
<td>accumulator</td>
<td></td>
<td>105</td>
</tr>
</tbody>
</table>
function f(a, b, c) {
    var local = c - 100;
    return a + local * b;
}

LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>[a]</td>
</tr>
<tr>
<td>a1</td>
<td>[b]</td>
</tr>
<tr>
<td>a2</td>
<td>[c]</td>
</tr>
<tr>
<td>r0</td>
<td>[local]</td>
</tr>
<tr>
<td>accumulator</td>
<td></td>
</tr>
</tbody>
</table>
Ignition Bytecode Pipeline

Parser → Abstract Syntax Tree → Bytecode Generator
void BytecodeGenerator::VisitAddExpression(BinaryOperation* expr) {
    Register lhs = VisitForRegisterValue(expr->left());
    VisitForAccumulatorValue(expr->right());
    builder()->AddOperation(lhs);
}
void BytecodeGenerator::VisitObjectLiteral(ObjectLiteral* expr) {
  // Copy the literal boilerplate.
  int fast_clone_properties_count = 0;
  if (FastCloneShallowObjectStub::IsSupported(expr)) {
    STATIC_ASSERT(
        FastCloneShallowObjectStub::kMaximumClonedProperties <=
        1 << CreateObjectLiteralFlags::FastClonePropertiesCountBits::kShift);
    fast_clone_properties_count =
        FastCloneShallowObjectStub::PropertiesCount(expr->properties_count());
  }
  uint8_t flags =
      CreateObjectLiteralFlags::FlagsBits::encode(expr->ComputeFlags()) |
      CreateObjectLiteralFlags::FastClonePropertiesCountBits::encode(
          fast_clone_properties_count);
  builder()->CreateObjectLiteral(expr->constant_properties(),
      expr->literal_index(), flags);
  // Allocate in the outer scope since this register is used to return the
  // expression's results to the caller.
  Register literal = register_allocator()->outer()->NewRegister();
  builder()->StoreAccumulatorInRegister(literal);
  // Store computed values into the literal.
  int property_index = 0;
  AccessorTable accessor_table(zone());
  for (; property_index < expr->properties()->length(); property_index++) {
    ObjectLiteral::Property* property = expr->properties()->at(property_index);
    if (property->is_computed_name()) break;
    if (property->isCompileTimeValue()) continue;
    RegisterAllocationScope inner_register_scope(this);
    Literal* literal_key = property->key()->AsLiteral();
    switch (property->kind()) {
      case ObjectLiteral::Property::CONSTANT:
        UNREACHABLE();
        case ObjectLiteral::Property::MATERIALIZED_LITERAL:
          DCHECK(!CompileTimeValue::IsCompileTimeValue(property->value()));
          // Fall through.
          case ObjectLiteral::Property::COMPUTED:
            // It is safe to use [[Put]] here because the boilerplate already
            // contains computed properties with an uninitialized value.
            if (literal_key->value()->IsInternalizedString()) {
              if (property->emit_store()) {
                VisitForAccumulatorValue(property->value());
                if (FunctionLiteral::NeedsHomeObject(property->value())) {
                  RegisterAllocationScope register_scope(this);
                  Register value = register_allocator()->NewRegister();
                  builder()->StoreNamedProperty(
                      literal, literal_key->AsPropertyName(),
                      feedback_index(property->GetSlot(0)), language_mode());
                  VisitSetHomeObject(value, literal, property, 1);
                } else {
                  builder()->StoreNamedProperty(
                      literal, literal_key->AsPropertyName(),
                      feedback_index(property->GetSlot(0)), language_mode());
                }
              } else {
                builder()->VisitForEffect(property->value());
              }
            } else {
              builder()->VisitForEffect(property->value());
            }
        break;
    }
  }
  register_allocator()->PrepareForConsecutiveAllocations(4);
  Register literal_argument =
      register_allocator()->NextConsecutiveRegister();
  Register key = register_allocator()->NextConsecutiveRegister();
  Register value = register_allocator()->NextConsecutiveRegister();
  Register language = register_allocator()->NextConsecutiveRegister();
  builder()->MoveRegister(literal, literal_argument);
  builder()->StoreAccumulatorInRegister(key);
  builder()->StoreAccumulatorInRegister(value);
  if (property->emit_store()) {
    builder()->LoadLiteral(Smi::FromInt(SLOPPY)).StoreAccumulatorInRegister(language).
        CallRuntime(Runtime::kSetProperty, literal_argument, 4);
    VisitSetHomeObject(value, literal, property, 4);
  }
  builder()->PrepareForConsecutiveAllocations(2);
  builder()->NextConsecutiveRegister();
  builder()->NextConsecutiveRegister();
  builder()->NextConsecutiveRegister();
  builder()->NextConsecutiveRegister();
}

Ignition Bytecode Pipeline
builder()->MoveRegister(literal, literal_argument);
builder()->StoreAccumulatorInRegister(literal_argument);
for (AccessorTable::Iterator it = accessor_table.begin();
    it != accessor_table.end(); ++it) {
  Register AllocationScope inner_register_scope(this);
  register_allocator()->PrepareForConsecutiveAllocations(5);
  Register literal_argument = register_allocator()->NextConsecutiveRegister();
  Register key = register_allocator()->NextConsecutiveRegister();
  Register value = register_allocator()->NextConsecutiveRegister();
  Register attr = register_allocator()->NextConsecutiveRegister();
  builder()->MoveRegister(literal, literal_argument);
  builder()->StoreAccumulatorInRegister(name);
  VisitObjectLiteralAccessor(literal, it->second->getter, getter);
  builder()->VisitObjectLiteralAccessor(literal, it->second->setter, setter);
  Register allocator = register_allocator()->NextConsecutiveRegister();
  Register literal = register_allocator()->NextConsecutiveRegister();
  Register allocator = register_allocator()->NextConsecutiveRegister();
  builder()->MoveRegister(literal, literal_argument);
  VisitForAccumulatorValue(property->value());
  builder()->StoreAccumulatorInRegister(allocator);
  builder()->CallRuntime(Runtime::kInternalSetPrototype, literal_argument, 2);
  case ObjectLiteral::Property::GETTER:
    if (property->emit_store()) {
      accessor_table.lookup(literal_key)->second->getter = property;
    }
    break;
  case ObjectLiteral::Property::SETTER:
    if (property->emit_store()) {
      accessor_table.lookup(literal_key)->second->setter = property;
    }
    break;
  } // Define accessors, using only a single call to the runtime for each pair of
  // corresponding getters and setters.
  for (AccessorTable::Iterator it = accessor_table.begin();
    it != accessor_table.end(); ++it) {
    Register AllocationScope inner_register_scope(this);
    register_allocator()->PrepareForConsecutiveAllocations(5);
    Register literal_argument = register_allocator()->NextConsecutiveRegister();
    Register key = register_allocator()->NextConsecutiveRegister();
    Register value = register_allocator()->NextConsecutiveRegister();
    Register attr = register_allocator()->NextConsecutiveRegister();
    builder()->MoveRegister(literal, literal_argument);
    VisitObjectLiteralAccessor(literal, it->second->getter, getter);
    builder()->VisitObjectLiteralAccessor(literal, it->second->setter, setter);
    builder()->LoadLiteral(Smi::FromInt(NONE)).StoreAccumulatorInRegister(attr);
    builder()->CallRuntime(Runtime::kDefineDataPropertyInLiteral, literal_argument, 2);
    continue;
  }
Ignition Bytecode Pipeline

Parser → Abstract Syntax Tree → Bytecode Generator
Ignition Bytecode Pipeline

- Parser
- Abstract Syntax Tree
- Bytecode Generator
- Register Optimizer
- Peephole Optimizer
- Dead-code Elimination
Ignition Bytecode Pipeline

- Parser
- Abstract Syntax Tree
- Bytecode Generator
- Register Optimizer
- Peephole Optimizer
- Dead-code Elimination
- Bytecode Array Writer
- Bytecode
Ignition Bytecode Pipeline

Parser → Abstract Syntax Tree → Bytecode Generator → Register Optimizer → Peephole Optimizer → Dead-code Elimination → Bytecode Array Writer → Bytecode → Interpreter
Building the Ignition Interpreter

- Write in C++
Building the Ignition Interpreter

❌ Write in C++
- Need trampolines between Interpreted and JITed functions
- Can’t interoperate with fast code-stubs
Building the Ignition Interpreter

❌ Write in C++
  - Need trampolines between Interpreted and JITed functions
  - Can’t interoperate with fast code-stubs

  – Hand-crafted assembly code
Building the Ignition Interpreter

❌ Write in C++
  ○ Need trampolines between Interpreted and JITed functions
  ○ Can’t interoperate with fast code-stubs

❌ Hand-crafted assembly code
  ○ Would need to be ported to 9 architectures
Building the Ignition Interpreter

❌ Write in C++
  ○ Need trampolines between Interpreted and JITed functions
  ○ Can’t interoperate with fast code-stubs

❌ Hand-crafted assembly code
  ○ Would need to be ported to 9 architectures
  – Backend of the TurboFan Compiler
Building the Ignition Interpreter

❌ Write in C++
  ○ Need trampolines between Interpreted and JITed functions
  ○ Can’t interoperate with fast code-stubs

❌ Hand-crafted assembly code
  ○ Would need to be ported to 9 architectures

✔ Backend of the TurboFan Compiler
  ○ Write-once in macro-assembly
  ○ Architecture specific instruction selection optimizations for free
  ○ Relatively painless interoperability with existing code-stubs
TurboFan Pipeline

- JavaScript Source
- JavaScript
- Simple
- Machine
- Scheduler
- CodeGen

Machine Code
TurboFan Pipeline

- JavaScript Source
- JavaScript
- Simple
- Machine
- Scheduler
- CodeGen
- WebAssembly

Interpreters:
- Machine Code
- Interpreted Assembler
Building an Interpreter using TurboFan

```c
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
    Node* reg_index = assembler->BytecodeOperandReg(0);
    Node* lhs = assembler->LoadRegister(reg_index);
    Node* rhs = assembler->GetAccumulator();
    Node* result = AddStub::Generate(assembler, lhs, rhs);
    assembler->SetAccumulator(result);
    assembler->Dispatch();
}
```
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
    Node* reg_index = assembler->BytecodeOperandReg(0);
    Node* lhs = assembler->LoadRegister(reg_index);
    Node* rhs = assembler->GetAccumulator();
    Node* result = AddStub::Generate(assembler, lhs, rhs);
    assembler->SetAccumulator(result);
    assembler->Dispatch();
}

~375 LOC for number addition
Building an Interpreter using TurboFan

```c++
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
    Node* reg_index = assembler->BytecodeOperandReg(0);
    Node* lhs = assembler->LoadRegister(reg_index);
    Node* rhs = assembler->GetAccumulator();
    Node* result = AddStub::Generate(assembler, lhs, rhs);
    assembler->SetAccumulator(result);
    assembler->Dispatch();
}
```

~375 LOC for number addition
~250 LOC for string addition
Building an Interpreter using TurboFan

```cpp
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
    Node* reg_index = assembler->BytecodeOperandReg(0);
    Node* lhs = assembler->LoadRegister(reg_index);
    Node* rhs = assembler->GetAccumulator();
    Node* result = AddStub::Generate(assembler, lhs, rhs);
    assembler->SetAccumulator(result);
    assembler->Dispatch();
}
```

~375 LOC for number addition
~250 LOC for string addition
... for type conversions
## Indirect Threaded Bytecode Dispatch

<table>
<thead>
<tr>
<th>Ldar:</th>
<th>0x32e3e920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star:</td>
<td>0x32e3e9a0</td>
</tr>
<tr>
<td>Add:</td>
<td>0x32e400e0</td>
</tr>
<tr>
<td>Sub:</td>
<td>0x32e401e0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Indirect Threaded Bytecode Dispatch

Ldar:
- Ldar: 0x32e3e920
- Star: 0x32e3e9a0
- Add: 0x32e400e0
- Sub: 0x32e401e0

...
Indirect Threaded Bytecode Dispatch

<table>
<thead>
<tr>
<th></th>
<th>Ldar: 0x32e3e920</th>
<th>Star: 0x32e3e9a0</th>
<th>Add: 0x32e400e0</th>
<th>Sub: 0x32e401e0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>movsxbq rax,[r14+r12*1+0x1]</td>
<td>movq rax,[rbp+rax*8]</td>
<td>leaq r12,[r12+0x2]</td>
<td>movzxbl rbx,[r12+r14*1]</td>
</tr>
<tr>
<td></td>
<td>movq rbx,[r15+rbx*8]</td>
<td>jmp rbx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load operand #0
Indirect Threaded Bytecode Dispatch

Ldar: 0x32e3e920
Star: 0x32e3e9a0
Add: 0x32e400e0
Sub: 0x32e401e0
...

movsxbq rax,[r14+r12*1+0x1]
movq rax,[rbp+rax*8]
leaq r12,[r12+0x2]
movzxbl rbx,[r12+r14*1]
movq rbx,[r15+rbx*8]
jmp rbx

Load operand #0
Load register to accumulator
Indirect Threaded Bytecode Dispatch

Ldar:

Star: 0x32e3e9a0
Add: 0x32e400e0
Sub: 0x32e401e0

Load operand #0
Load register to accumulator
Advance to next bytecode

movsxbq rax,[r14+r12*1+0x1]
movq rax,[rbp+rax*8]
lea r12,[r12+0x2]
movzxb1 rbx,[r12+r14*1]
movq rbx,[r15+rbx*8]
jmp rbx
Indirect Threaded Bytecode Dispatch

<table>
<thead>
<tr>
<th>Ldar:</th>
<th>0x32e3e920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star:</td>
<td>0x32e3e9a0</td>
</tr>
<tr>
<td>Add:</td>
<td>0x32e400e0</td>
</tr>
<tr>
<td>Sub:</td>
<td>0x32e401e0</td>
</tr>
</tbody>
</table>

Ldar

Load operand #0
Load register to accumulator
Advance to next bytecode
Load next bytecode

movsxbq rax,[r14+r12*1+0x1]
movq rax,[rbp+rax*8]
leaq r12,[r12+0x2]
movzxb1 rbx,[r12+r14*1]
movq rbx,[r15+rbx*8]
jmp rbx
## Indirect Threaded Bytecode Dispatch

### Ldar

<table>
<thead>
<tr>
<th>Ldar</th>
<th>Star</th>
<th>Add</th>
<th>Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x32e3e920</td>
<td>0x32e3e9a0</td>
<td>0x32e400e0</td>
<td>0x32e401e0</td>
</tr>
</tbody>
</table>

...  

- **movsxbq rax,[r14+r12*1+0x1]**
- **movq rax,[rbp+rax*8]**
- **lea r12,[r12+0x2]**
- **movzxb1 rbx,[r12+r14*1]**
- **movq rbx,[r15+rbx*8]**
- **jmp rbx**

Load operand #0
Load register to accumulator
Advance to next bytecode
Load next bytecode
Dispatch
Inline Caches with Code Patching

Machine Code → IC Stubs
Inline Caches with Code Patching

Machine Code

IC Stubs
Inline Caches with Type Feedback Vector

Machine Code  Feedback Vector  IC Stubs
Inline Caches with Type Feedback Vector

Bytecode → Feedback Vector → IC Stubs
Inline Caches with Type Feedback Vector

Bytecode  Feedback Vector  IC Stubs
Ignition vs Full-Codegen

Octane (Nexus 5)
Crankshaft and TurboFan disabled
Ignition vs Full-Codegen

Octane (Nexus 5)
Crankshaft and TurboFan disabled
Ignition vs Default

Score

Octane Score (x64)
Ignition vs Default

40% slower

5% slower
Ignition vs Default

Tune Tiering up

Octane Score (x64)

Score

40% slower

5% slower
Ignition vs Default

Tune Tiering up

OSR

40% slower

Octane Score (x64)

5% slower
Ignition vs Default

Tune Tiering up
OSR
BinaryOp Feedback

40% slower

5% slower
Ignition vs Default

- Tuning Tiering up
- OSR
- BinaryOp Feedback
- CS / TF adaptation

Octane Score (x64)

Score

40% slower

5% slower
Real Websites

Google Maps

LinkedIn

Facebook
Real Websites
Real Websites
Summary
Summary

- JavaScript is hard
Summary

● JavaScript is hard
● V8 is complex
Summary

- JavaScript is hard
- V8 is complex
- An interpreter can (sometimes) beat a JIT...
Summary

- JavaScript is hard
- V8 is complex
- An interpreter can (sometimes) beat a JIT... but it takes a lot of work!
Ignition Bytecodes

Loading the accumulator
LdaZero
LdaSmi8
LdaUndefined
LdrUndefined
LdaNull
LdaTheHole
LdaTrue
LdaFalse
LdaConstant

Binary Operators
Add
Sub
Mul
Div
Mod
BitwiseOr
BitwiseXor
BitwiseAnd
ShiftLeft
ShiftRight
ShiftRightLogical

Closure Allocation
CreateClosure

Globals
LdaGlobal
LdrGlobal
LdaGlobalInsideTypeof
StaGlobalSloppy
StaGlobalStrict

Unary Operators
Inc
Dec
LogicalNot
TypeOf
DeletePropertyStrict
DeletePropertySloppy

Call Operations
Call
TailCall
CallRuntime
CallRuntimeForPair
CallJsRuntime
InvokeIntrinsic

New Operator
New

Test Operators
TestEqual
TestNotEqual
TestEqualStrict
TestLessThan
TestGreaterThan
TestLessThanOrEqual
TestGreaterThanOrEqual
TestInstanceOf
TestIn

Context Operations
PushContext
PopContext
LdaContextSlot
LdrContextSlot
StaContextSlot

Cast Operators
ToName
ToNumber
ToObject

Arguments Allocation
CreateMappedArguments
CreateUnmappedArguments
CreateRestParameter

Register Transfers
Ldar
Star
Mov

Control Flow
Jump
JumpConstant
JumpIfTrue
JumpIfTrueConstant
JumpIfFalse
JumpIfFalseConstant
JumpIfToBooleanTrue
JumpIfToBooleanTrueConstant
JumpIfToBooleanFalse
JumpIfToBooleanFalseConstant
JumpIfNull
JumpIfNullConstant
JumpIfUndefined
JumpIfUndefinedConstant
JumpIfNotHole
JumpIfNotHoleConstant

Load Property Operations
LdaNamedProperty
LdaKeyedProperty
KeyedLoadICStrict

Store Property Operations
StoreICSloppy
StoreICStrict
KeyedStoreICSloppy
KeyedStoreICStrict

Complex Flow Control
ForInPrepare
ForInNext
ForInDone
ForInStep

Generators
SuspendGenerator
ResumeGenerator

Literals
CreateRegExpLiteral
CreateArrayLiteral
CreateObjectLiteral

Loaders
CreateRegExpLiteral
CreateArrayLiteral
CreateObjectLiteral

Control Flow
Jump
JumpConstant
JumpIfTrue
JumpIfTrueConstant
JumpIfFalse
JumpIfFalseConstant
JumpIfToBooleanTrue
JumpIfToBooleanTrueConstant
JumpIfToBooleanFalse
JumpIfToBooleanFalseConstant
JumpIfNull
JumpIfNullConstant
JumpIfUndefined
JumpIfUndefinedConstant
JumpIfNotHole
JumpIfNotHoleConstant

Non-Local Flow Control
Throw
ReThrow
Return