Ignition: Jump-starting an Interpreter for V8

Ross McIlroy
Google London

Google

Agenda

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

Google

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

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

Something Simple

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

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

Integer addition

Google

Something Simple

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

add(1, 2);                  // 3

add(1.2, 3.14);             // 4.34
```

Integer addition

Floating point addition

Something Simple

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

add(1, 2);                  // 3
add(1.2, 3.14);             // 4.34
add("hello", "world");      // "helloworld"
```

Integer addition

Floating point addition

String addition

Something Simple

```
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
```

Integer addition

Floating point addition

String addition

Type coercion

Something Simple

```
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"
```

Integer addition

Floating point addition

String addition

Type coercion

Something Simple

```
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**

operator +

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 *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be 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 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

operator +

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 *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be 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 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.

ToPrimitive

7.1.1 **ToPrimitive (input [, PreferredType])**

The abstract operation ToPrimitive takes an *input* argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

1. If *PreferredType* was not passed, let *hint* be `"default"`.
2. Else if *PreferredType* is hint String, let *hint* be `"string"`.
3. Else *PreferredType* is hint Number, let *hint* be `"number"`.
4. Let *exoticToPrim* be GetMethod(*input*, @@toPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. Let *result* be Call(*exoticToPrim*, *input*, «*hint*»).
    b. ReturnIfAbrupt(*result*).
    c. If Type(*result*) is not Object, return *result*.
    d. Throw a **TypeError** exception.
7. If *hint* is `"default"`, let *hint* be `"number"`.
8. Return OrdinaryToPrimitive(*input*,*hint*).

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either `"string"` or `"number"`.
3. If *hint* is `"string"`, then
    a. Let *methodNames* be «`"toString"`, `"valueOf"`».
4. Else,
    a. Let *methodNames* be «`"valueOf"`, `"toString"`».
5. For each *name* in *methodNames* in List order, do
    a. Let *method* be Get(*O*, *name*).
    b. ReturnIfAbrupt(*method*).
    c. If IsCallable(*method*) is **true**, then
        i. Let *result* be Call(*method*, *O*).
        ii. ReturnIfAbrupt(*result*).
        iii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

A Glance at Semantics

operator +

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 *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be 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 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.

ToString

7.1.12 ToString (argument)

The abstract operation ToString converts *argument* to a value of type String according to Table 12:

Table 12 — ToString Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]]).
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

1. If *m* is NaN, return the String `"NaN"`.
2. If *m* is +0 or −0, return the String `"0"`.
3. If *m* is less than zero, return the String concatenation of the String `"-"` and ToString(−*m*).
4. If *m* is +∞, return the String `"Infinity"`.
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the String consisting of the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes), followed by $n-k$ occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n \leq 21$, return the String consisting of the code units of the most significant *n* digits of the decimal representation of *s*, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-n$ digits of the decimal representation of *s*.
8. If $-6 < n \leq 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by $-n$ occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the *k* digits of the decimal representation of *s*.
9. Otherwise, if $k = 1$, return the String consisting of the code unit of the single digit of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of *s*, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-1$ digits of the decimal representation of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).

ToPrimitive

7.1.1 ToPrimitive (input [, PreferredType])

The abstract operation ToPrimitive takes an *input* argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

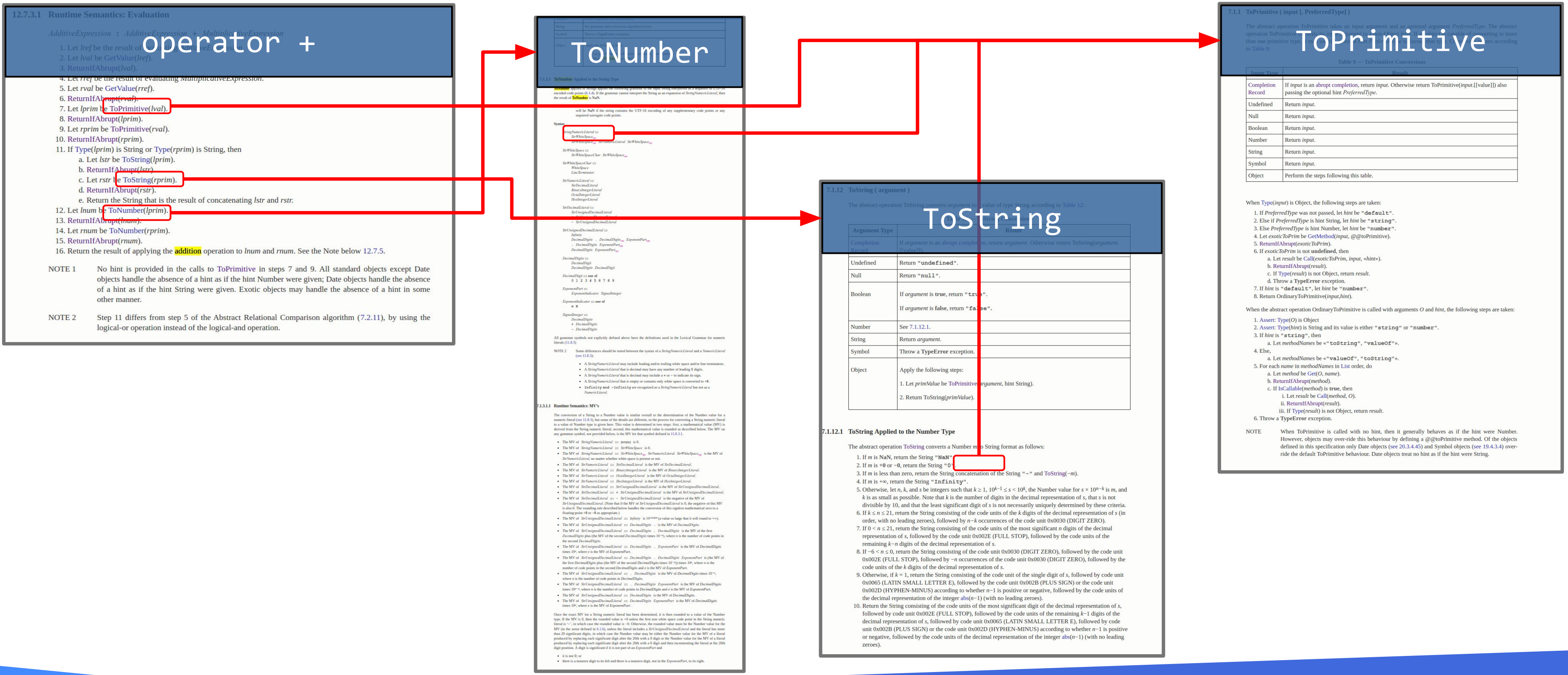
1. If *PreferredType* was not passed, let *hint* be `"default"`.
2. Else if *PreferredType* is hint String, let *hint* be `"string"`.
3. Else *PreferredType* is hint Number, let *hint* be `"number"`.
4. Let *exoticToPrim* be GetMethod(*input*, @@toPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. Let *result* be Call(*exoticToPrim*, *input*, «*hint*»).
    b. ReturnIfAbrupt(*result*).
    c. If Type(*result*) is not Object, return *result*.
    d. Throw a **TypeError** exception.
7. If *hint* is `"default"`, let *hint* be `"number"`.
8. Return OrdinaryToPrimitive(*input*,*hint*).

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

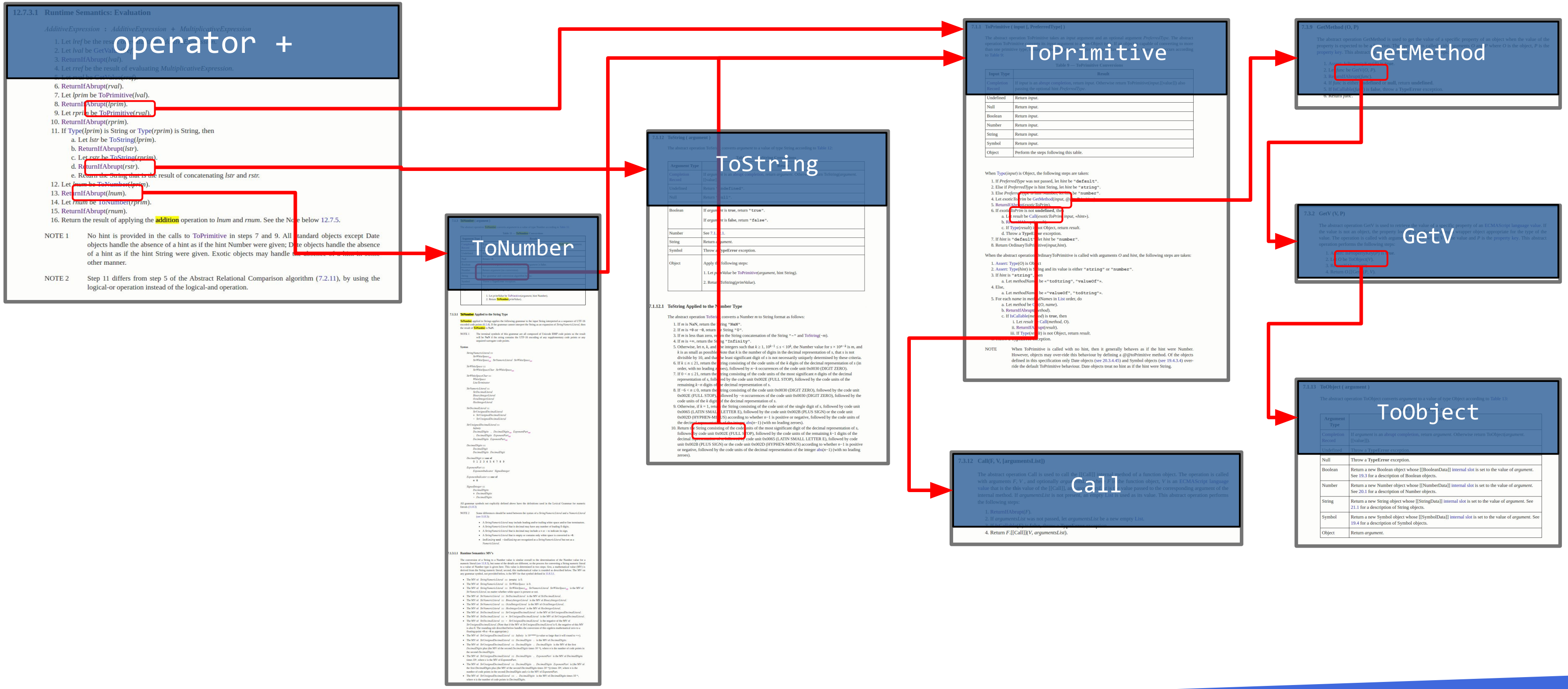
1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either `"string"` or `"number"`.
3. If *hint* is `"string"`, then
    a. Let *methodNames* be «`"toString"`, `"valueOf"`».
4. Else,
    a. Let *methodNames* be «`"valueOf"`, `"toString"`».
5. For each *name* in *methodNames* in List order, do
    a. Let *method* be Get(*O*, *name*).
    b. ReturnIfAbrupt(*method*).
    c. If IsCallable(*method*) is **true**, then
        i. Let *result* be Call(*method*, *O*).
        ii. ReturnIfAbrupt(*result*).
        iii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

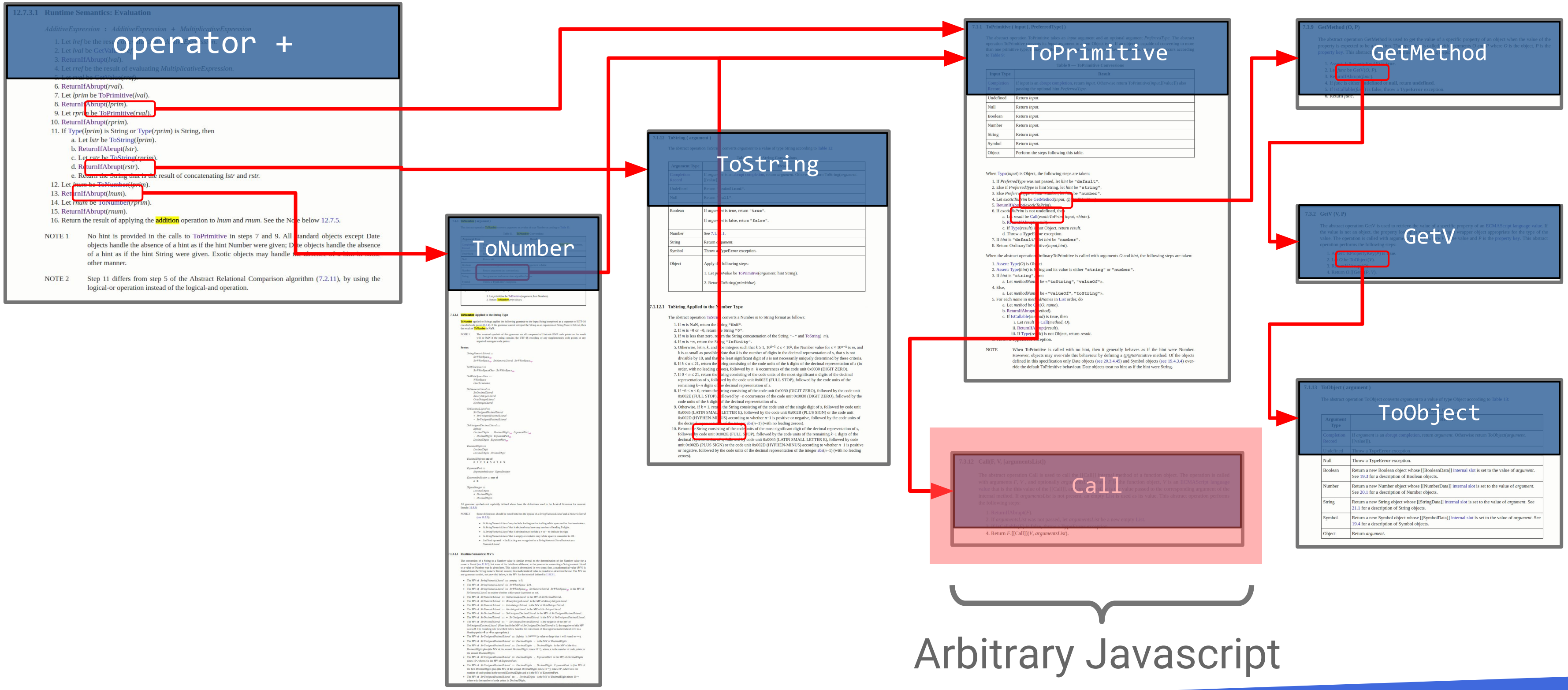
A Glance at Semantics



A Glance at Semantics

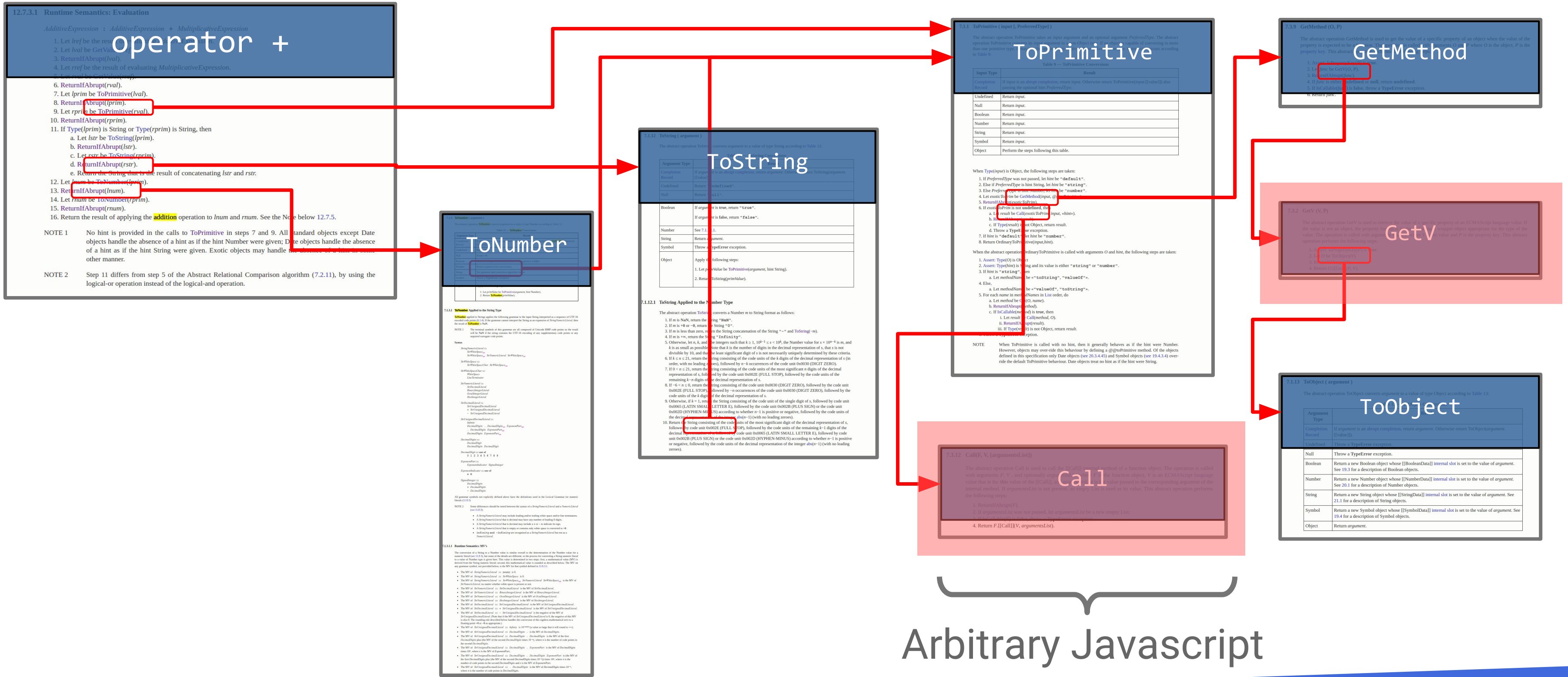


A Glance at Semantics



Google

A Glance at Semantics



Everything's a Function

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

An object's constructor is just a function

Everything's a Function

```
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

```
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

```
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

```
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);
tom.__proto__ = Object.prototype;
print(tom);                        // "[object Object]"
```

Which is completely dynamic....

Except when it’s a Closure

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

Except when it's a Closure

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

Closures over parameters, and mutable local variables

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

Google

Fun with eval()

```
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()

```
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()

```
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

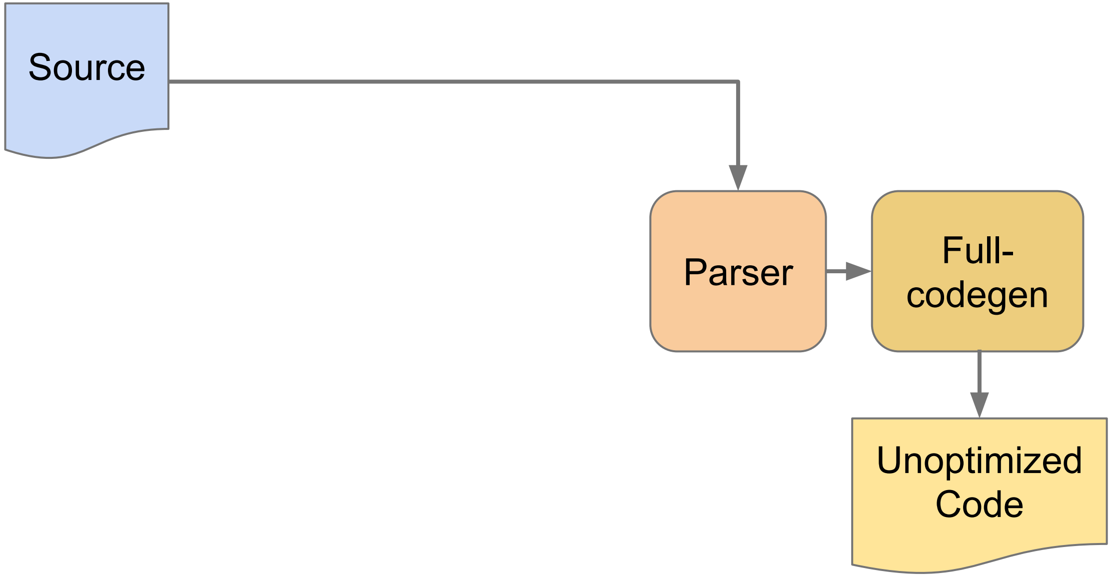
Or do crazy things...

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)

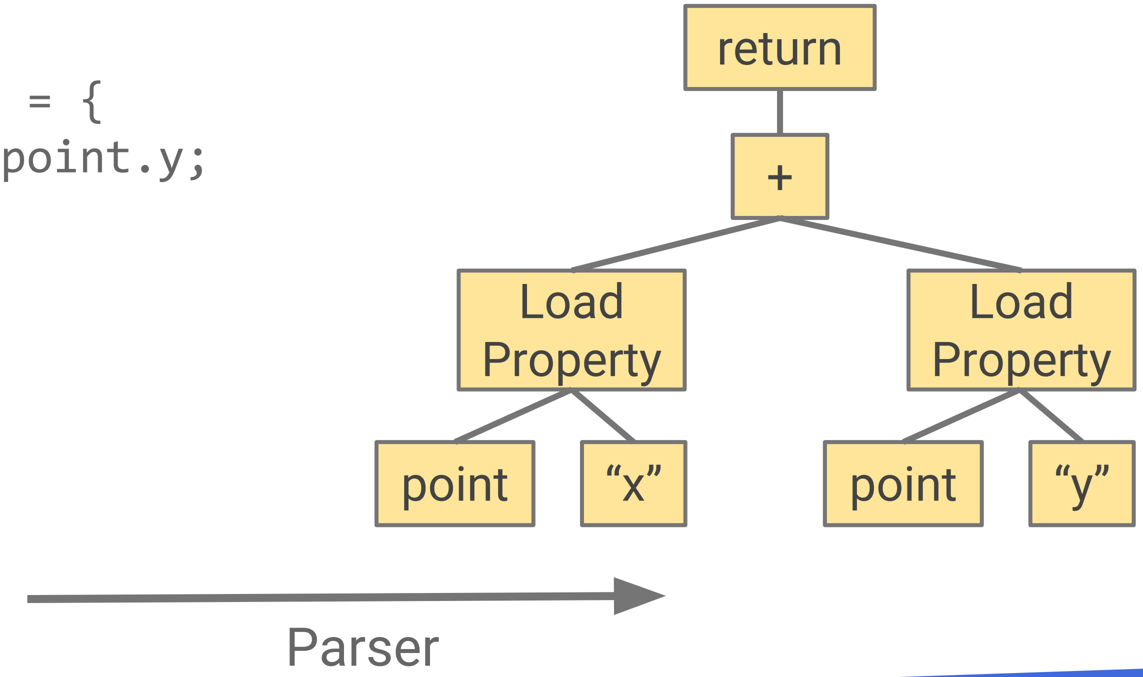


Full-Codegen in a nutshell

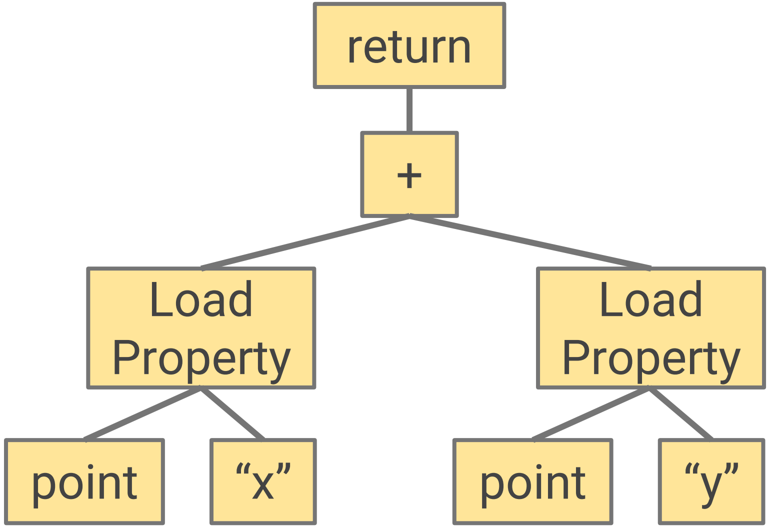
```
function Sum(point) = {
  return point.x + point.y;
};
```

Full-Codegen in a nutshell

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

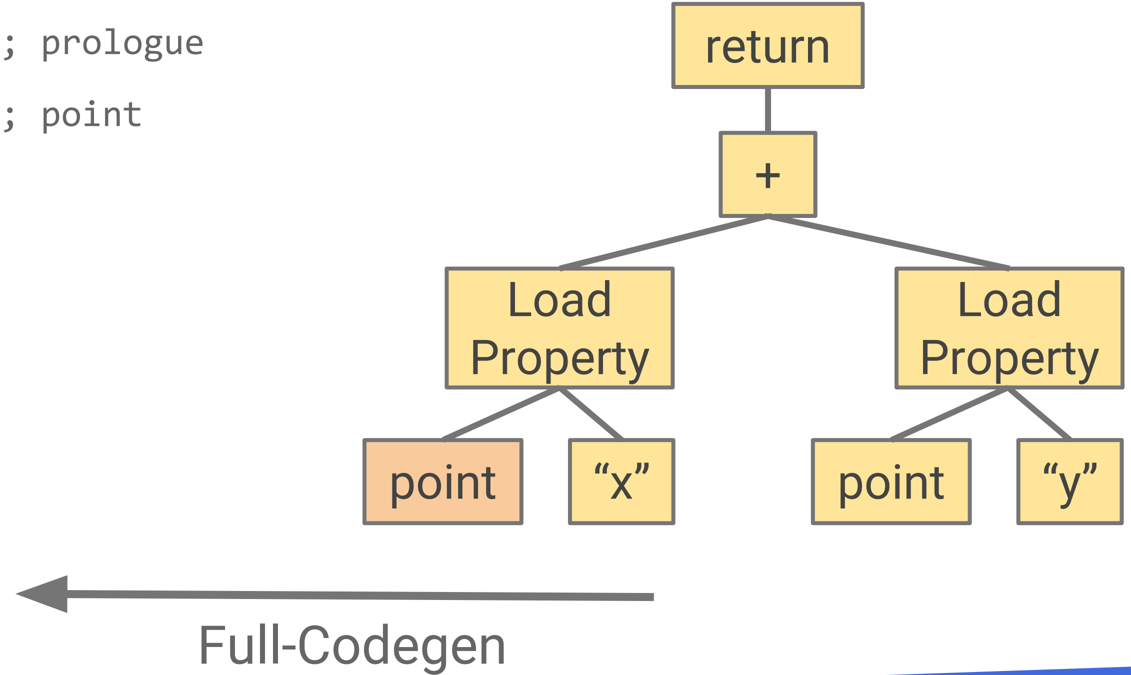


Full-Codegen in a nutshell



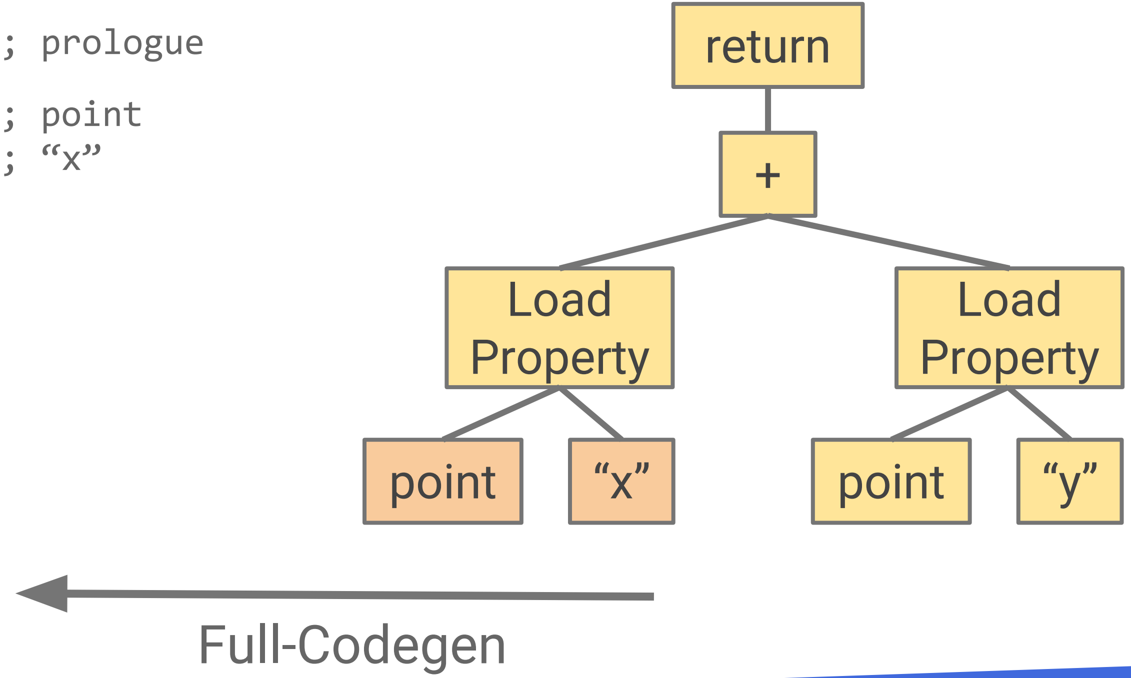
Full-Codegen in a nutshell

```
...                          ; prologue
mov eax, [ebp + 0x10]        ; point
```



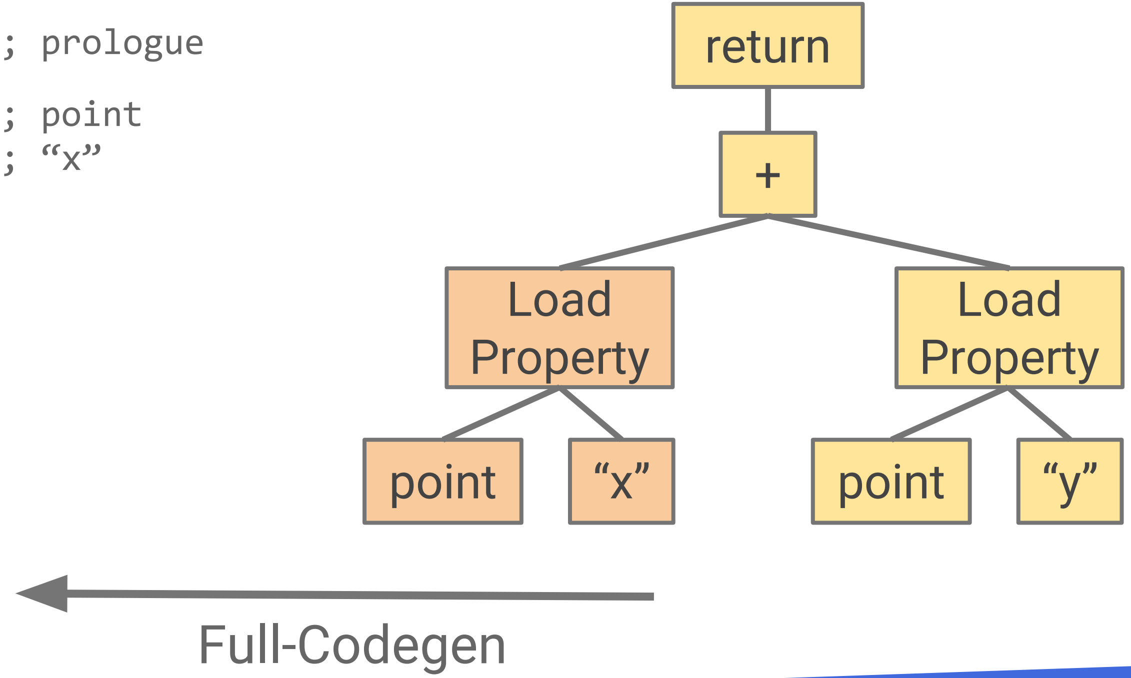
Full-Codegen in a nutshell

```
...                         ; prologue

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



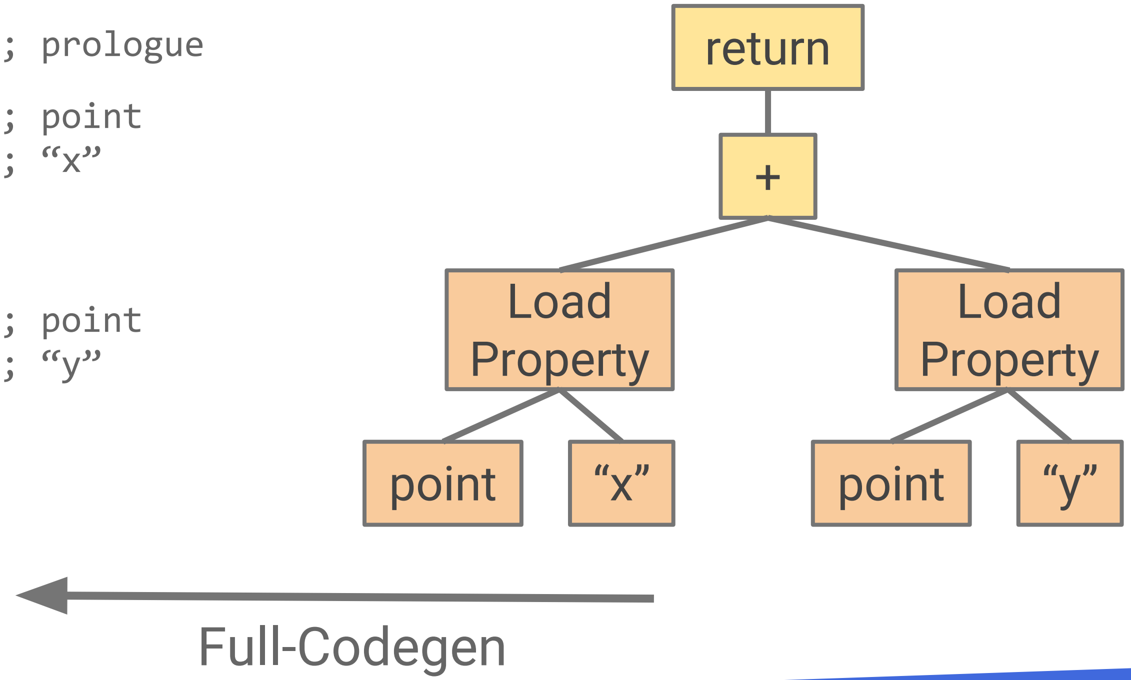
Full-Codegen in a nutshell

```
...                         ; prologue

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



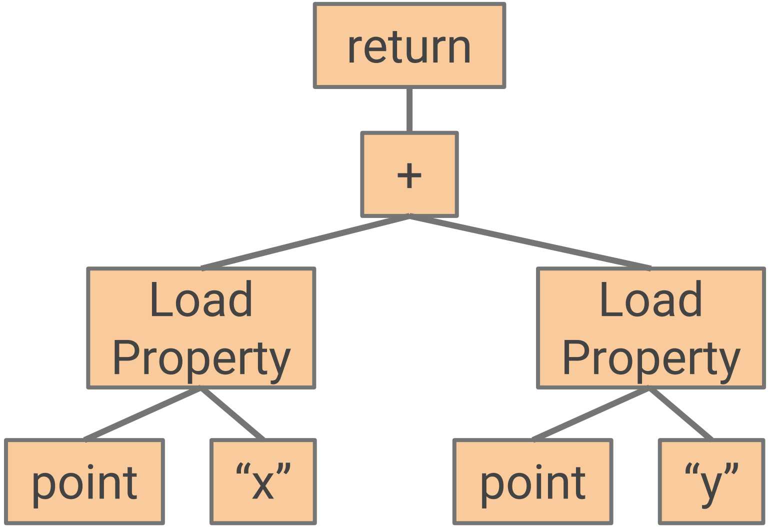
Full-Codegen in a nutshell

```
...                        ; 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
```



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

...
```

UNINITIALIZED_LOAD_IC

- ➔ Call into runtime
- ➔ Determine object layout
- ➔ Load property with <name>

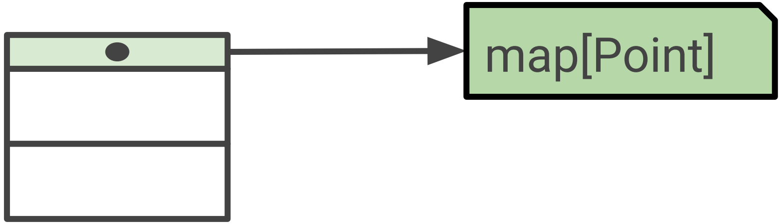
Hidden Classes

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

Hidden classes was a technique from Self VM

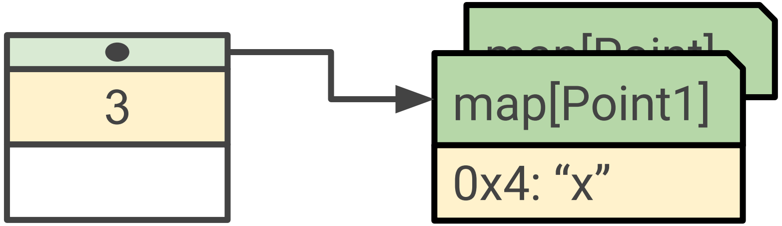
Hidden Classes

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

var point = new Point(3, 5);
```



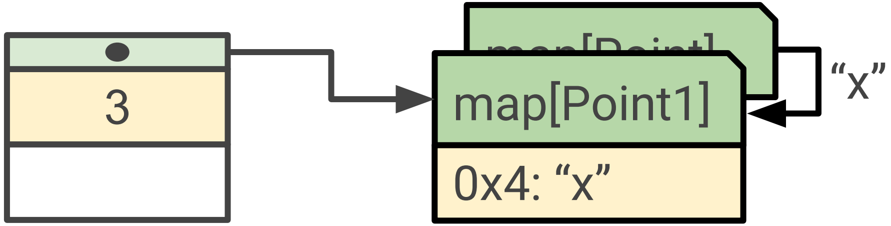
Hidden Classes

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

var point = new Point(3, 5);
```



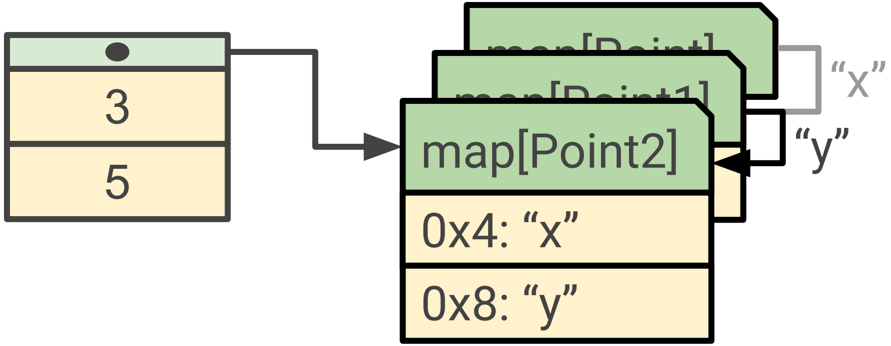
Hidden Classes

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

var point = new Point(3, 5);
```



Hidden Classes

```
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

...
```

MONOMORPHIC_LOAD_IC_X

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

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

...
```

```
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
```

Inline Caches (ICs)

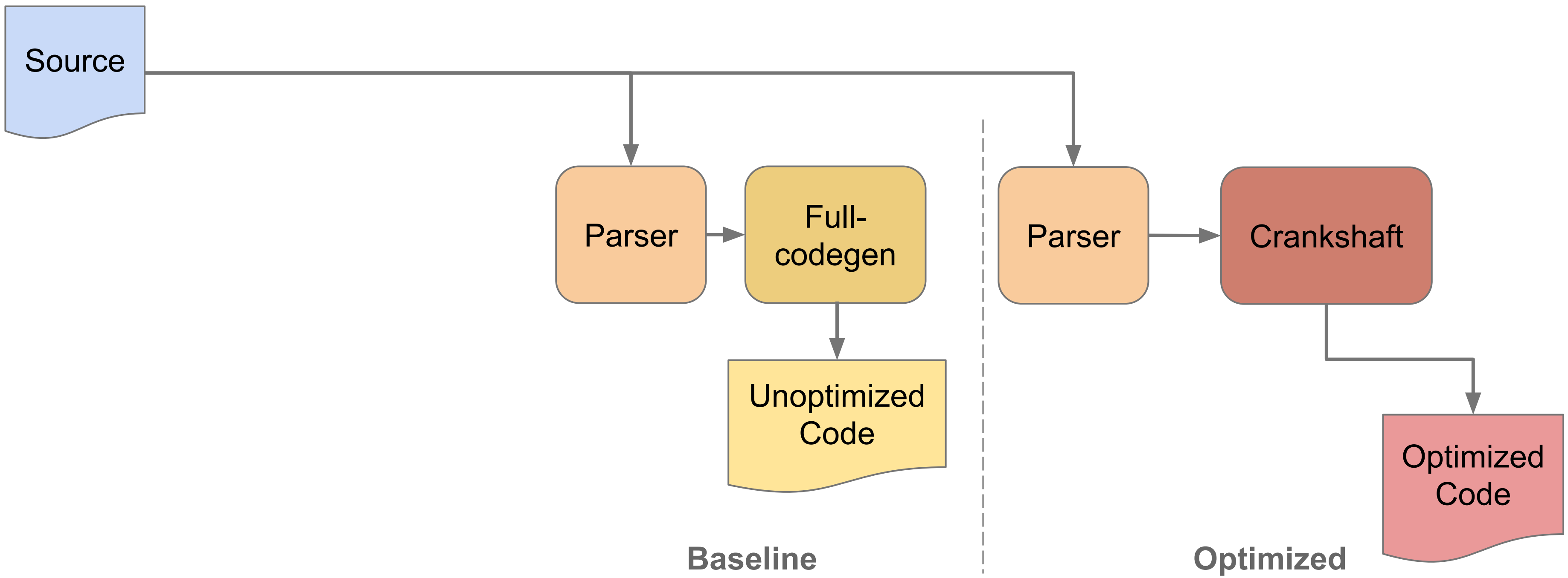
```
...                           ; prologue

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)



Google

A Little on Crankshaft

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

A Little on Crankshaft

```
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

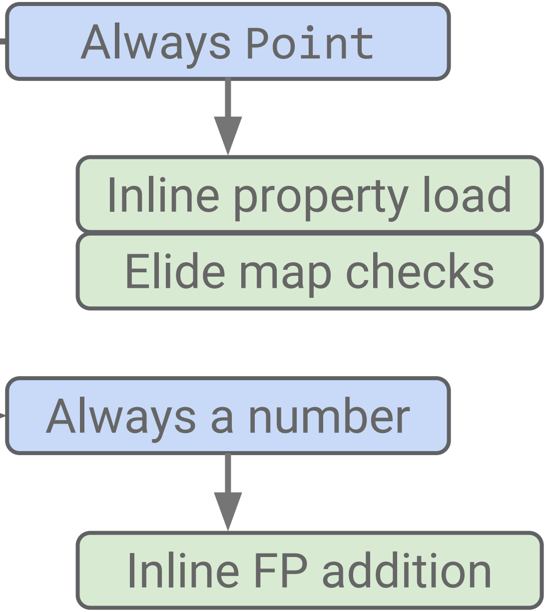
Always Point

```
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

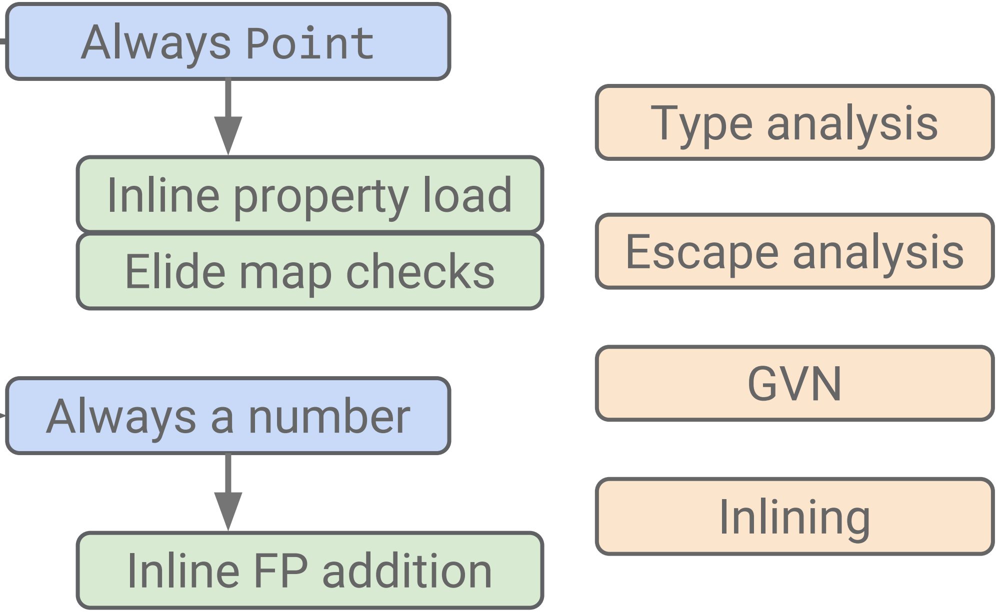
A Little on Crankshaft

```
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));
```



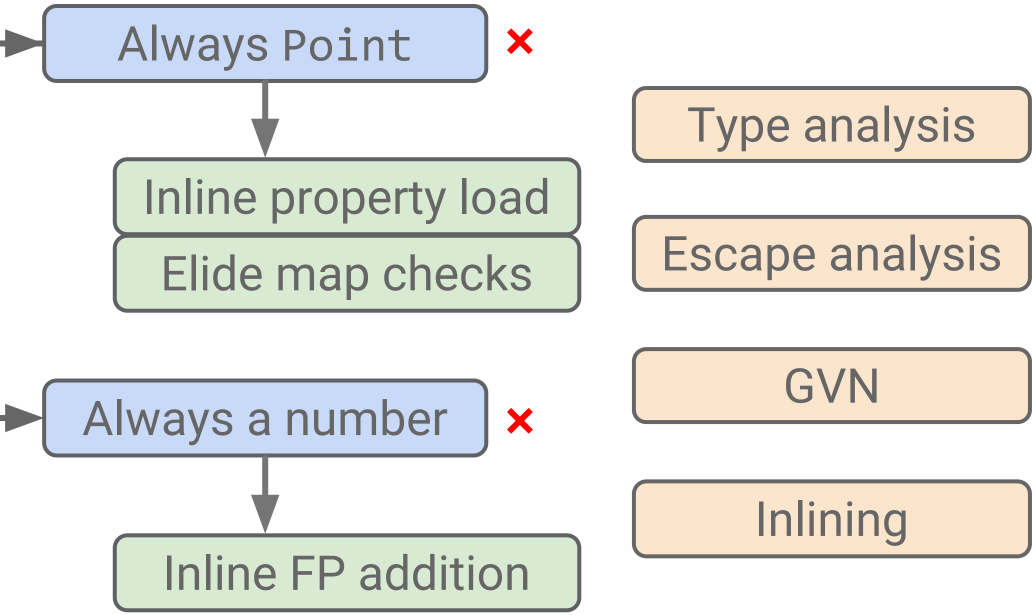
A Little on Crankshaft

```
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));
```



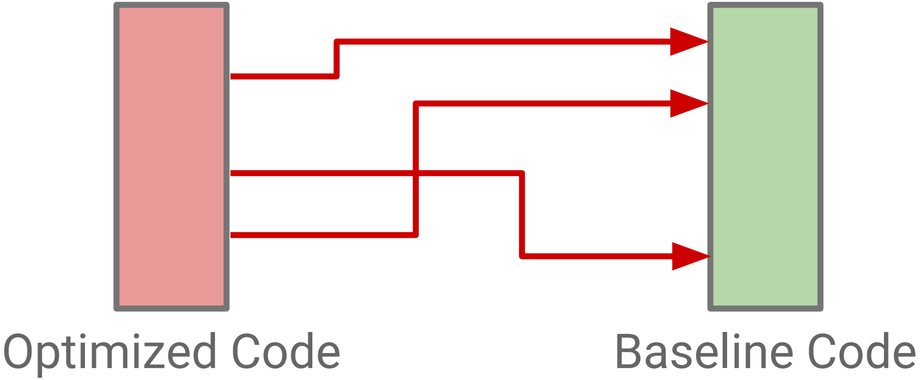
A Little on Crankshaft

```
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"));
```



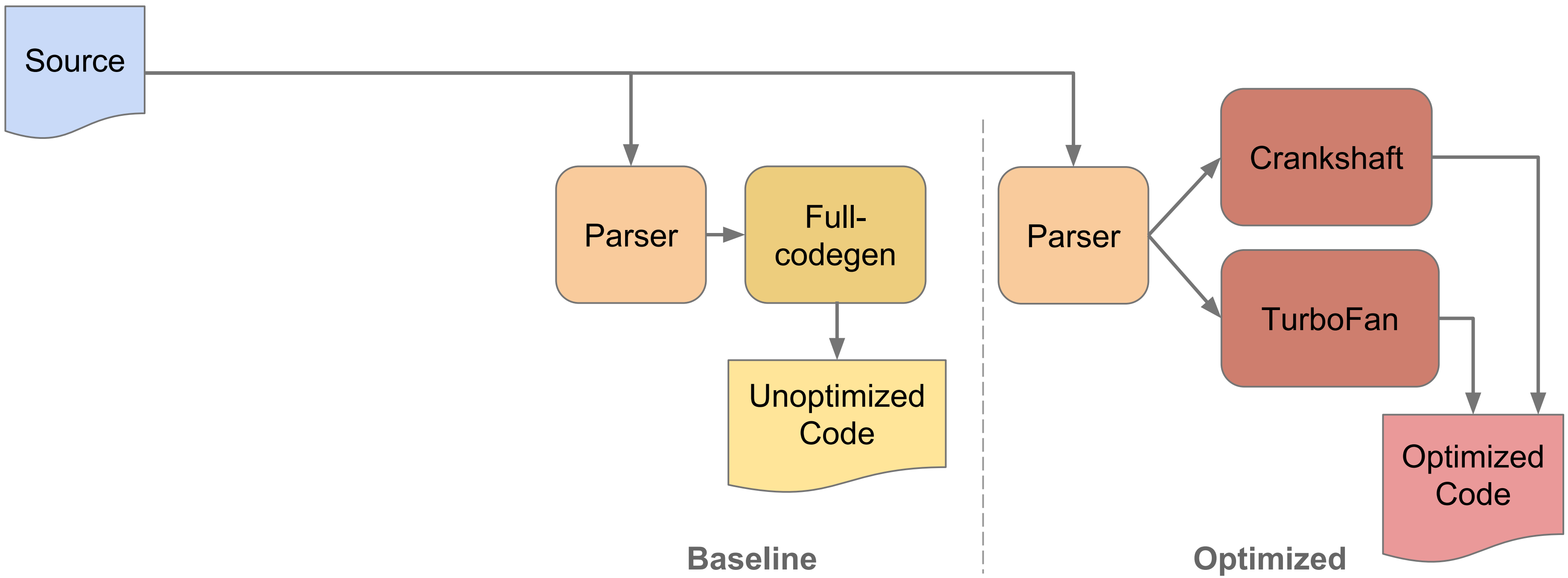
Deoptimization - Always Have a Backup Plan

- 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

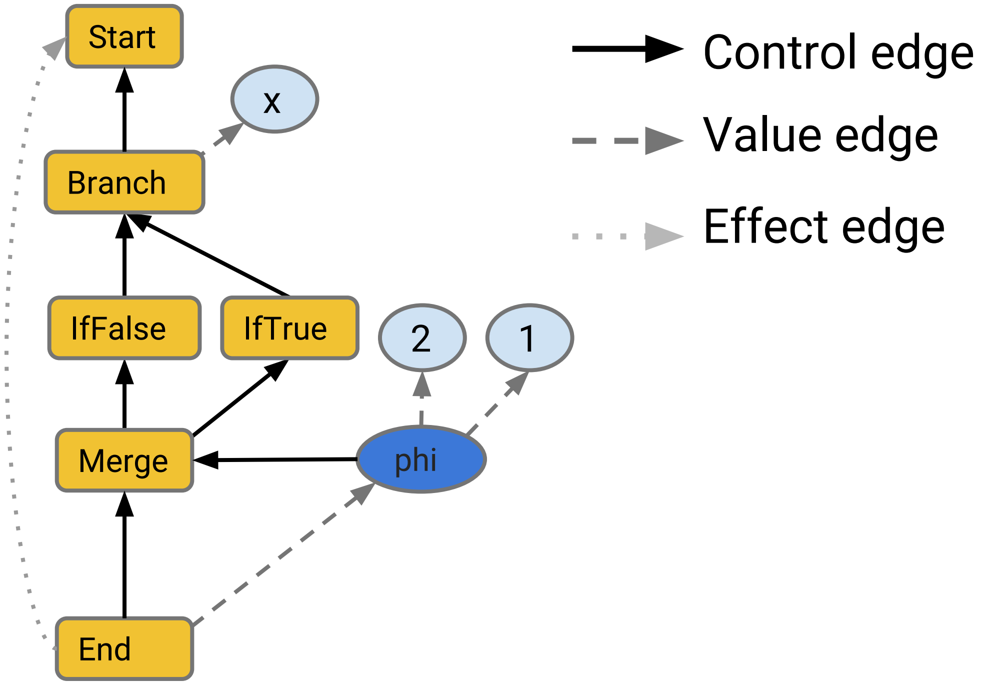
- Sea of Nodes
  - Relax evaluation order for most operations (value edges)
  - Skeleton of a CFG remains (control edges) and stateful operations (effect edges)
  - Provides better redundant code elimination and more code motion
- Three Level IR
  - JavaScript: JavaScript's overloaded operators
  - Simplified: 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

Sea of Nodes

```
function (x) {
  return x ? 1 : 2;
}
```

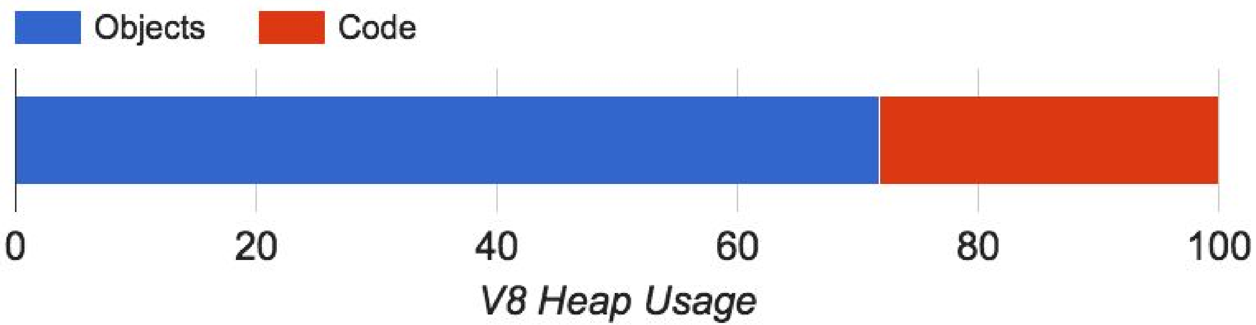


Retrofitting an Interpreter into a Moving Engine

Google

Why Interpret?

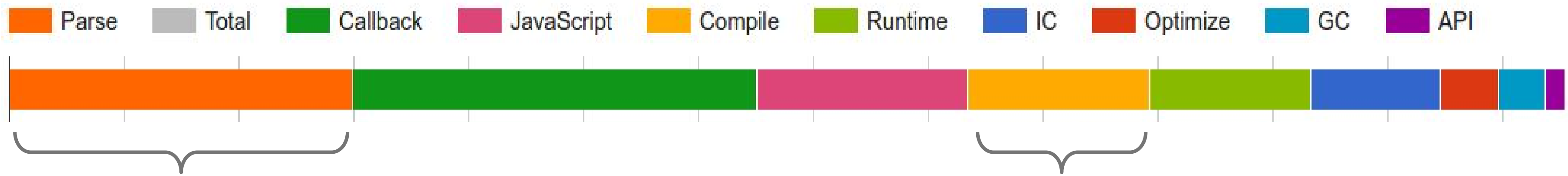
- Reduce memory usage



Why Interpret?

- Reduce memory usage

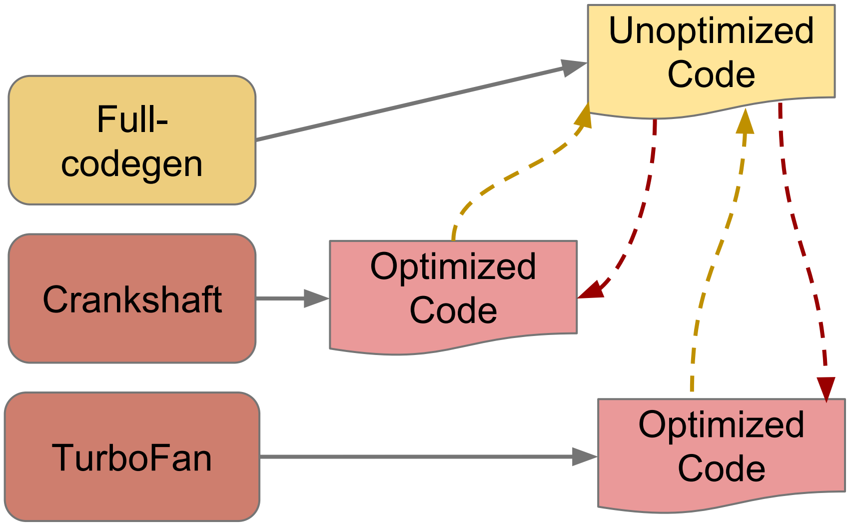
- Reduce startup time



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)

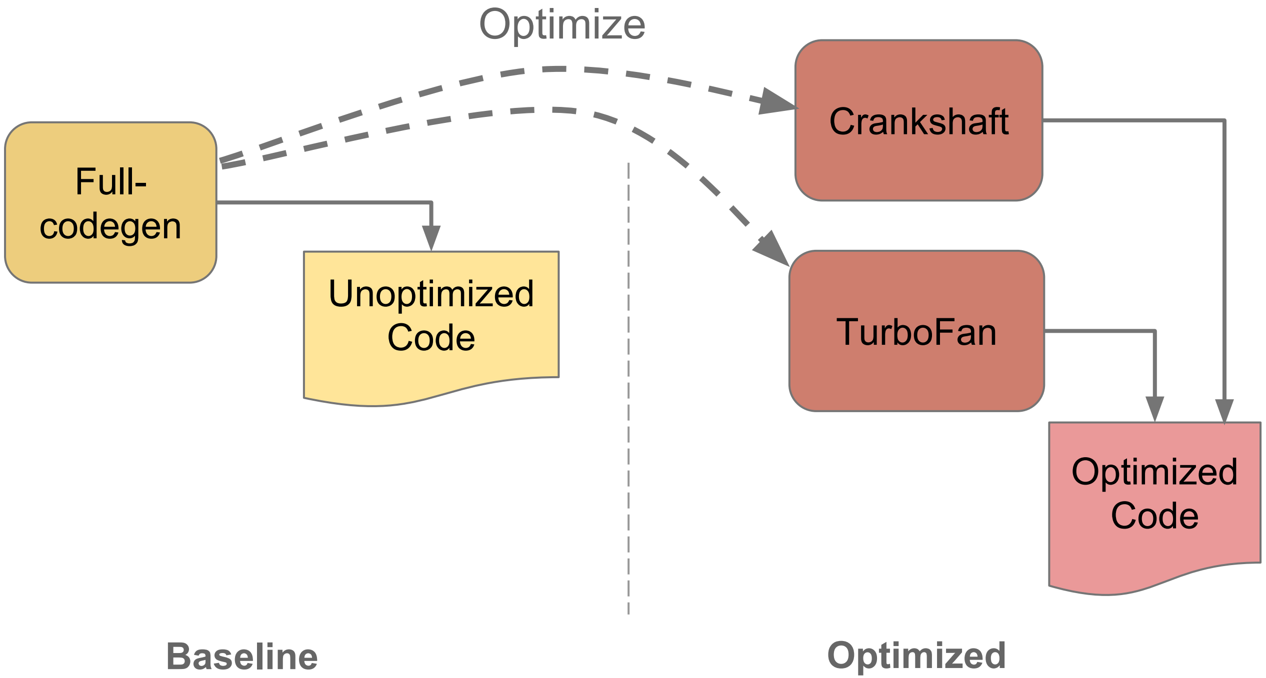
Ignition - Challenges

- 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

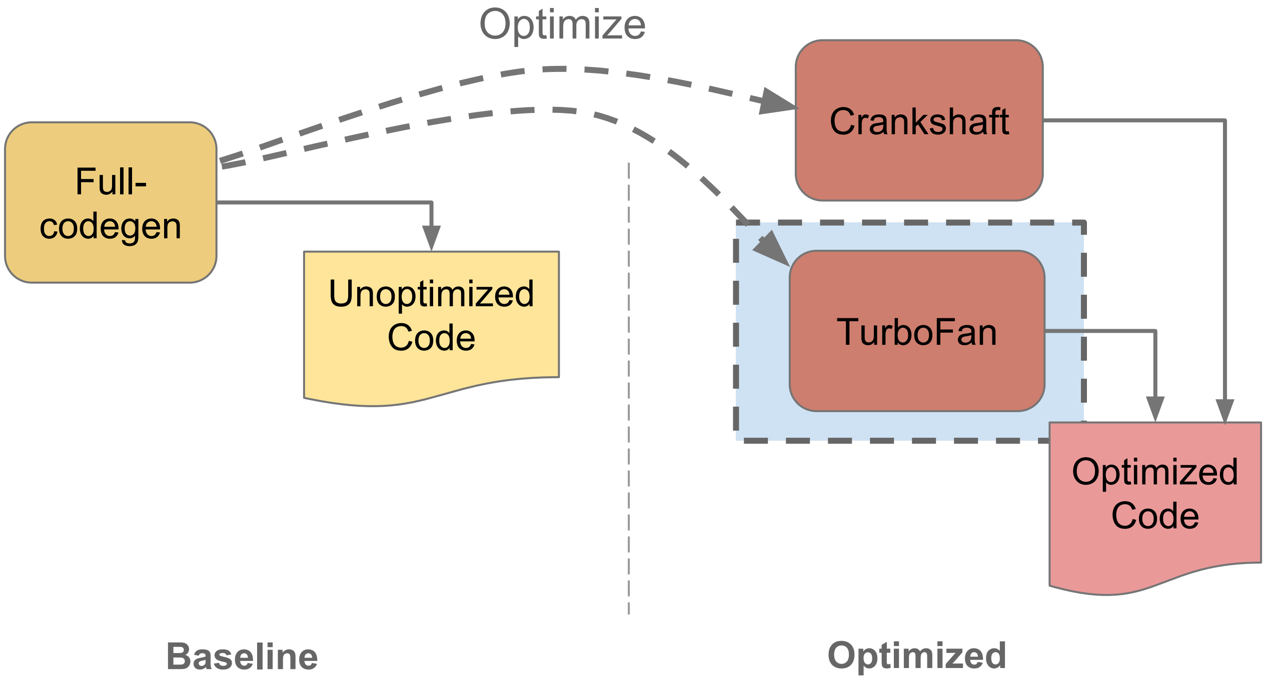
Ignition - Challenges

- 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)

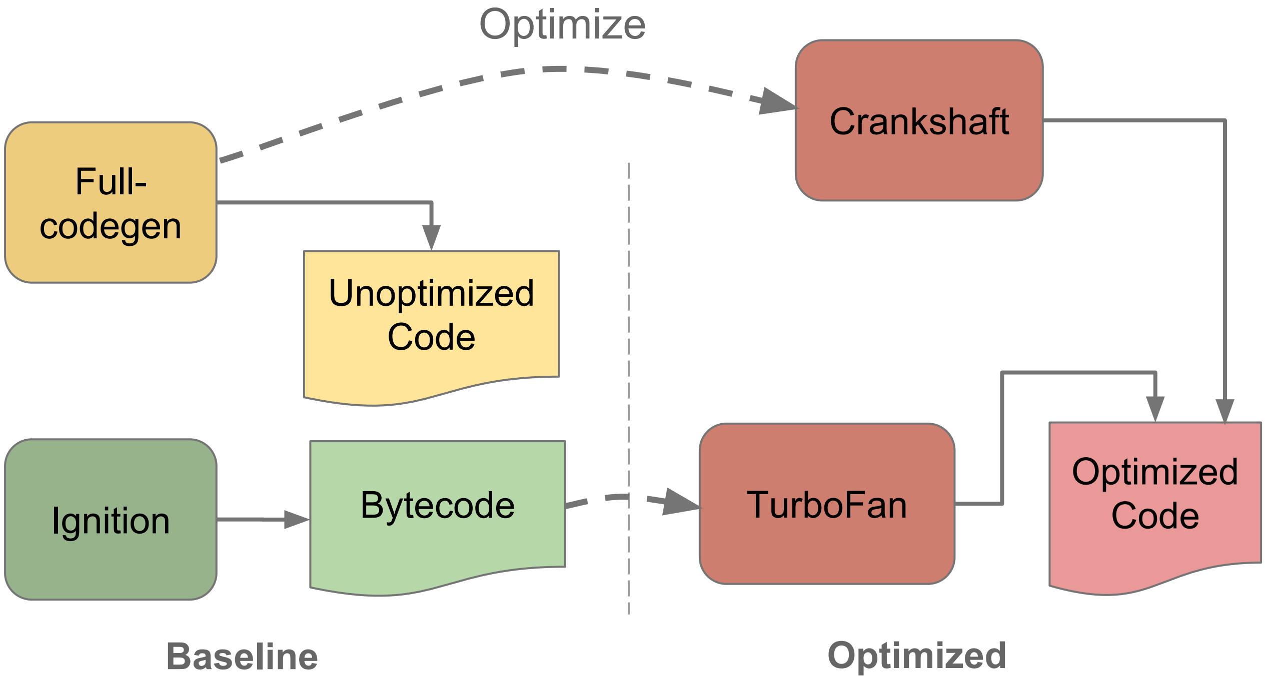


Compiler Pipeline (2015)

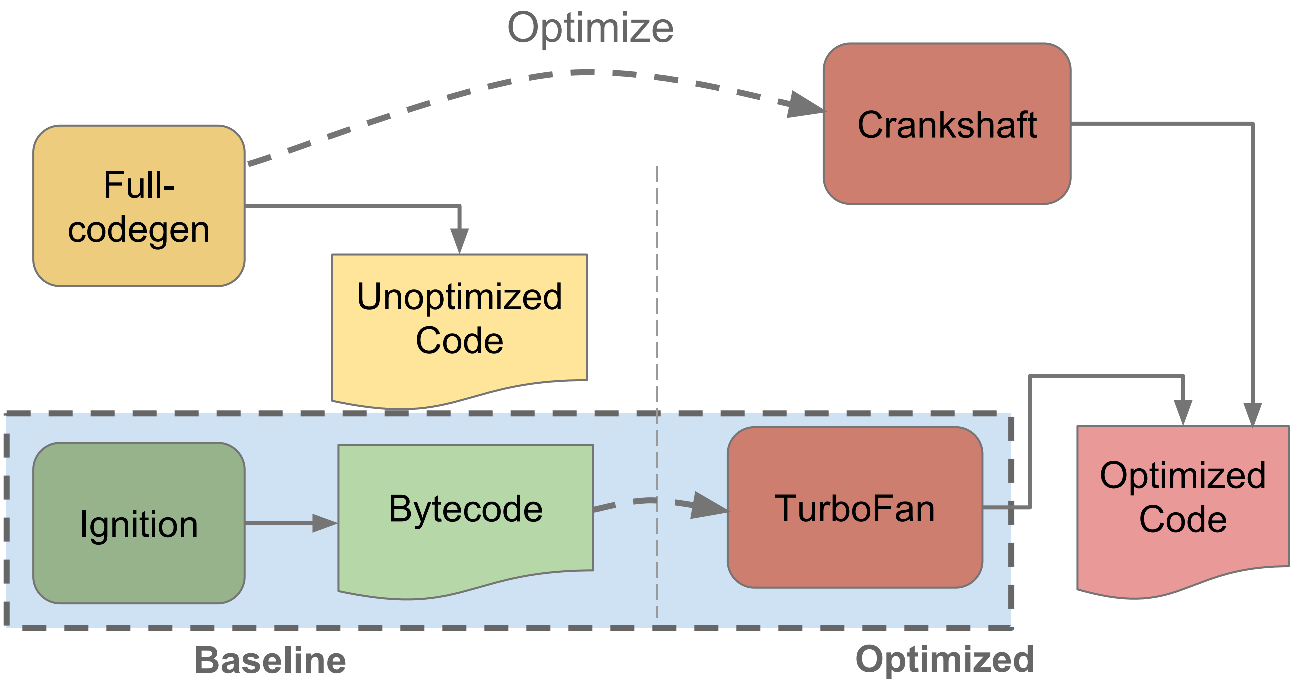


Google

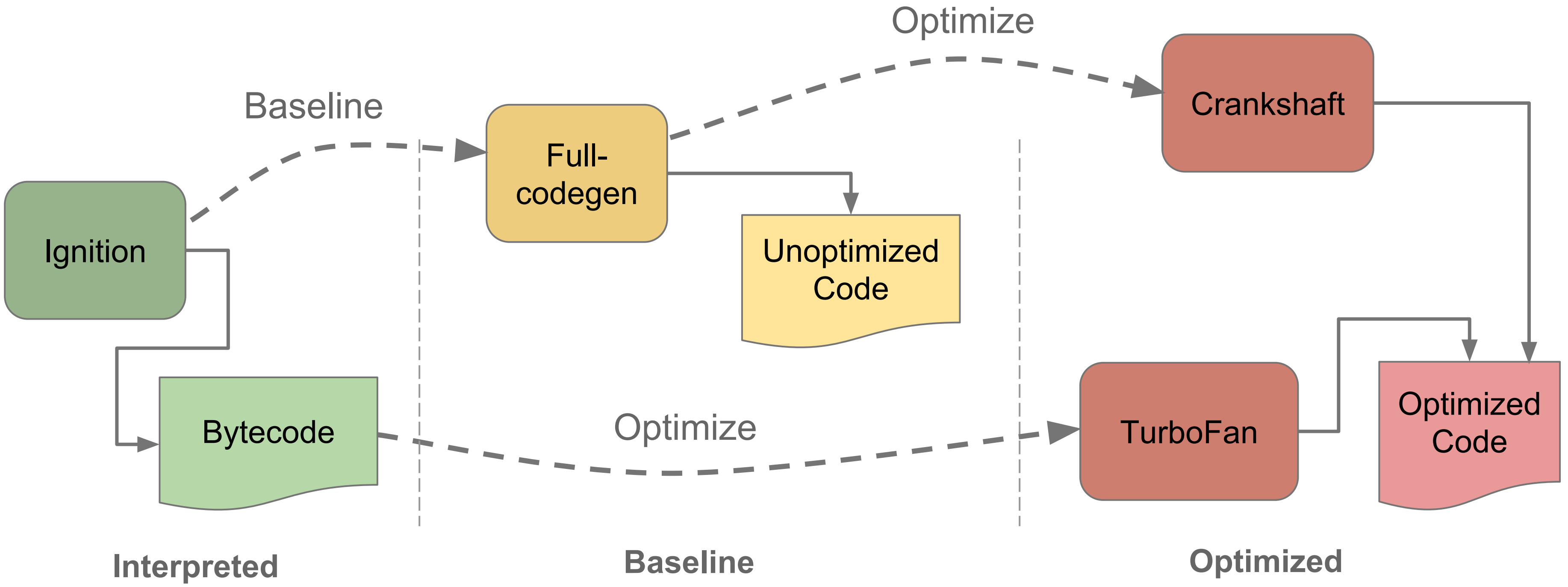
Compiler Pipeline (2016)



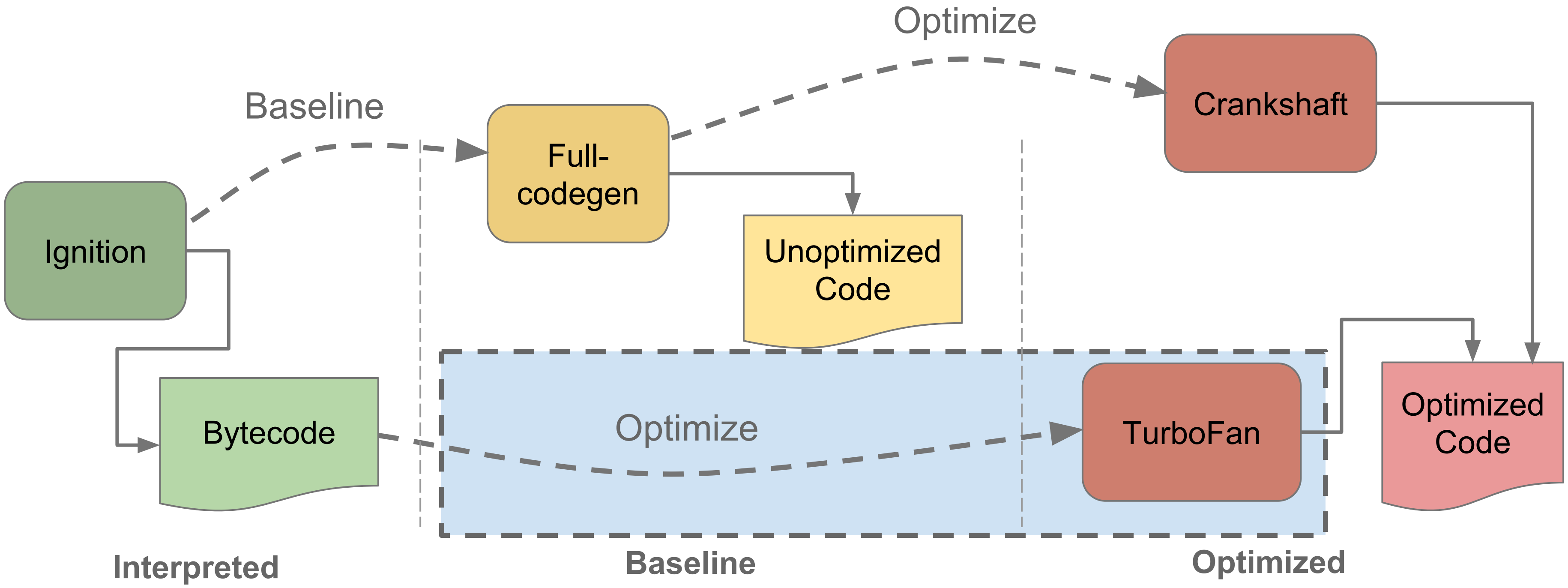
Compiler Pipeline (2016)



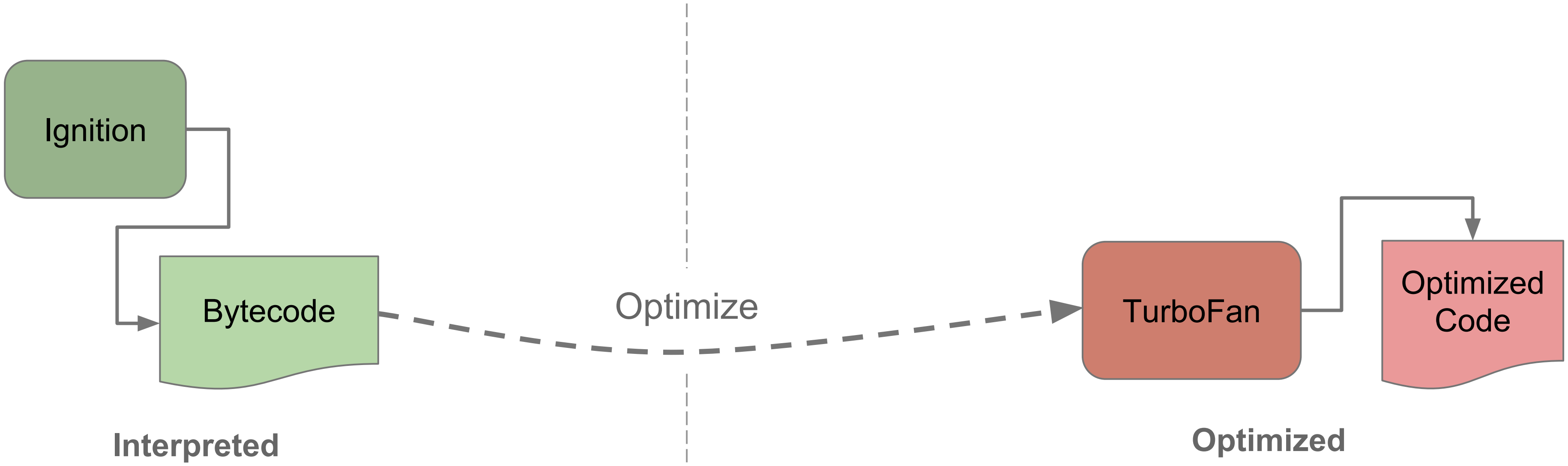
Compiler Pipeline (early 2017 ?)



Compiler Pipeline (early 2017 ?)



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

Ignition Bytecode

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

Ignition Bytecode

```
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

```
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
```

r0 [local]	undefined

Ignition Bytecode

```
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

Ignition Bytecode

```
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	undefined

Ignition Bytecode

```
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	100

Ignition Bytecode

```
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

Ignition Bytecode

```
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]	50
accumulator	50

Ignition Bytecode

```
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]	50
accumulator	2

Ignition Bytecode

```
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]	50
accumulator	100

Google

Ignition Bytecode

```
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]	50
accumulator	105

Google

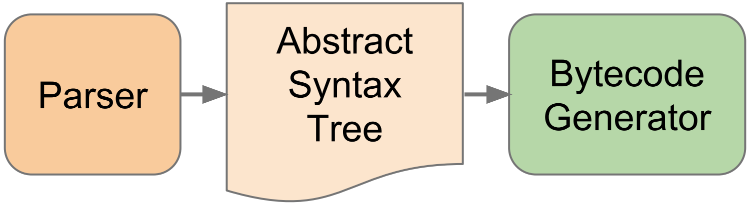
Ignition Bytecode

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

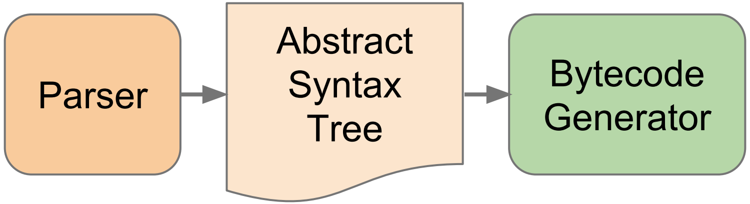
```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0 [a]	5
a1 [b]	2
a2 [c]	150
r0 [local]	50
accumulator	105

Ignition Bytecode Pipeline



Ignition Bytecode Pipeline




```
void BytecodeGenerator::VisitAddExpression(
    BinaryOperation* expr) {
  Register lhs =
      VisitForRegisterValue(expr->left());
  VisitForAccumulatorValue(expr->right());
  builder()->AddOperation(lhs);
}
```

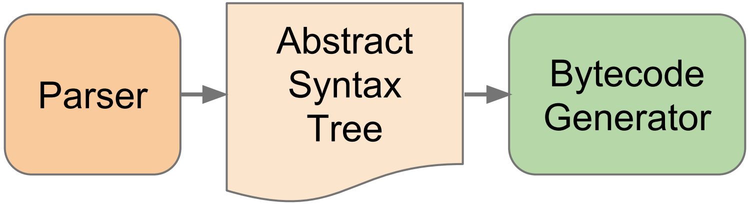
Ignition Bytecode Pipeline

```
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()->StoreAccumulatorInRegister(value);
              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 {
            VisitForEffect(property->value());
          }
        } else {
          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);
          VisitForAccumulatorValue(property->key());
          builder()->StoreAccumulatorInRegister(key);
          VisitForAccumulatorValue(property->value());
          builder()->StoreAccumulatorInRegister(value);
          if (property->emit_store()) {
            builder()
                ->LoadLiteral(Smi::FromInt(SLOPPY))
                .StoreAccumulatorInRegister(language)
                .CallRuntime(Runtime::kSetProperty, literal_argument, 4);
            VisitSetHomeObject(value, literal, property);
          }
        }
        break;
      }
      case ObjectLiteral::Property::PROTOTYPE: {
        DCHECK(property->emit_store());
        register_allocator()->PrepareForConsecutiveAllocations(2);
        Register literal_argument =
            register_allocator()->NextConsecutiveRegister();
        Register value = register_allocator()->NextConsecutiveRegister();
```

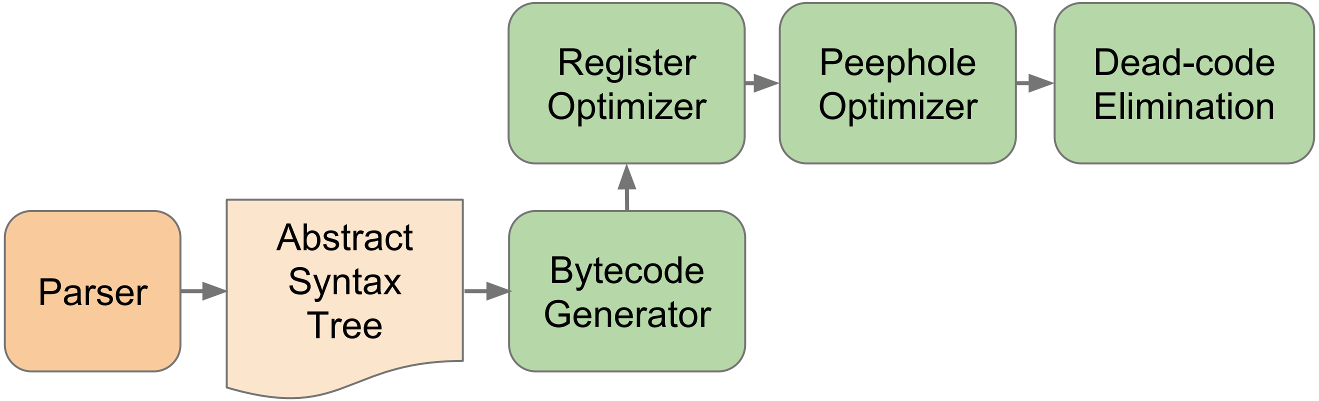
Ignition Bytecode Pipeline

```
builder()->MoveRegister(literal, literal_argument);
        VisitForAccumulatorValue(property->value());
        builder()->StoreAccumulatorInRegister(value).CallRuntime(
            Runtime::kInternalSetPrototype, literal_argument, 2);
        break;
      }
      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) {
    RegisterAllocationScope inner_register_scope(this);
    register_allocator()->PrepareForConsecutiveAllocations(5);
    Register literal_argument = register_allocator()->NextConsecutiveRegister();
    Register name = register_allocator()->NextConsecutiveRegister();
    Register getter = register_allocator()->NextConsecutiveRegister();
    Register setter = register_allocator()->NextConsecutiveRegister();
    Register attr = register_allocator()->NextConsecutiveRegister();
    builder()->MoveRegister(literal, literal_argument);
    VisitForAccumulatorValue(it->first);
    builder()->StoreAccumulatorInRegister(name);
    VisitObjectLiteralAccessor(literal, it->second->getter, getter);
    VisitObjectLiteralAccessor(literal, it->second->setter, setter);
    builder()
      ->LoadLiteral(Smi::FromInt(NONE))
      .StoreAccumulatorInRegister(attr)
      .CallRuntime(Runtime::kDefineAccessorPropertyUnchecked,
                   literal_argument, 5);
  }

  for (; property_index < expr->properties()->length(); property_index++) {
    ObjectLiteral::Property* property = expr->properties()->at(property_index);
    RegisterAllocationScope inner_register_scope(this);

    if (property->kind() == ObjectLiteral::Property::PROTOTYPE) {
      DCHECK(property->emit_store());
      register_allocator()->PrepareForConsecutiveAllocations(2);
      Register literal_argument =
          register_allocator()->NextConsecutiveRegister();
      Register value = register_allocator()->NextConsecutiveRegister();

      builder()->MoveRegister(literal, literal_argument);
      VisitForAccumulatorValue(property->value());
      builder()->StoreAccumulatorInRegister(value).CallRuntime(
          Runtime::kInternalSetPrototype, literal_argument, 2);
      continue;
    }
    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();
    DCHECK(Register::AreContiguous(literal_argument, key, value, attr));
    Register set_function_name =
        register_allocator()->NextConsecutiveRegister();

    builder()->MoveRegister(literal, literal_argument);
    VisitForAccumulatorValue(property->key());
    builder()->CastAccumulatorToName().StoreAccumulatorInRegister(key);
    VisitForAccumulatorValue(property->value());
    builder()->StoreAccumulatorInRegister(value);
    VisitSetHomeObject(value, literal, property);
    builder()->LoadLiteral(Smi::FromInt(NONE)).StoreAccumulatorInRegister(attr);
    switch (property->kind()) {
      case ObjectLiteral::Property::CONSTANT:
      case ObjectLiteral::Property::COMPUTED:
      case ObjectLiteral::Property::MATERIALIZED_LITERAL:
        builder()
            ->LoadLiteral(Smi::FromInt(property->NeedsSetFunctionName()))
            .StoreAccumulatorInRegister(set_function_name);
        builder()->CallRuntime(Runtime::kDefineDataPropertyInLiteral,
                               literal_argument, 5);
        break;
      case ObjectLiteral::Property::PROTOTYPE:
        UNREACHABLE();  // Handled specially above.
        break;
      case ObjectLiteral::Property::GETTER:
        builder()->CallRuntime(Runtime::kDefineGetterPropertyUnchecked,
                               literal_argument, 4);
        break;
      case ObjectLiteral::Property::SETTER:
        builder()->CallRuntime(Runtime::kDefineSetterPropertyUnchecked,
                               literal_argument, 4);
        break;
    }
  }
  execution_result()->SetResultInRegister(literal);
}
```

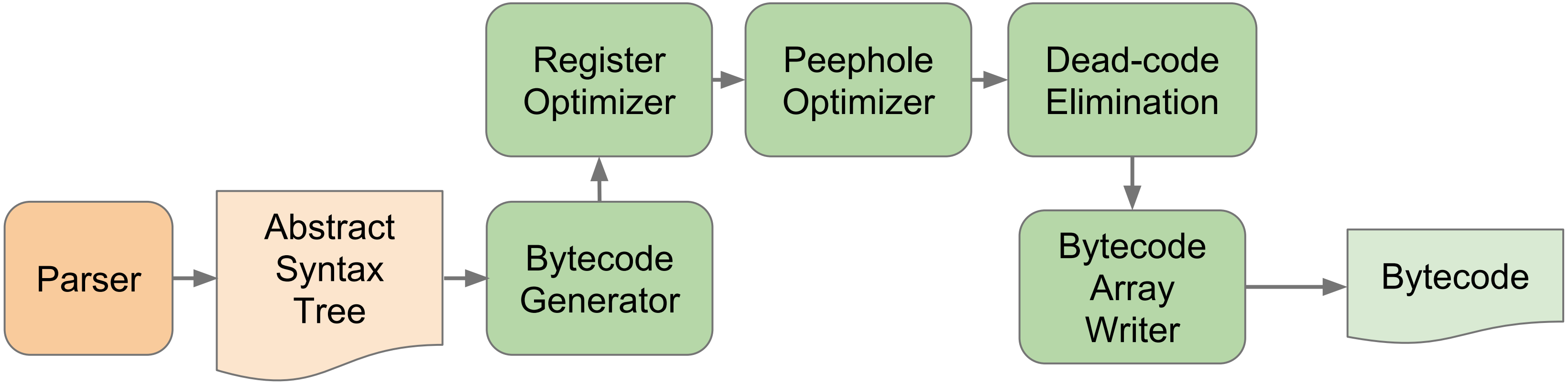
Ignition Bytecode Pipeline



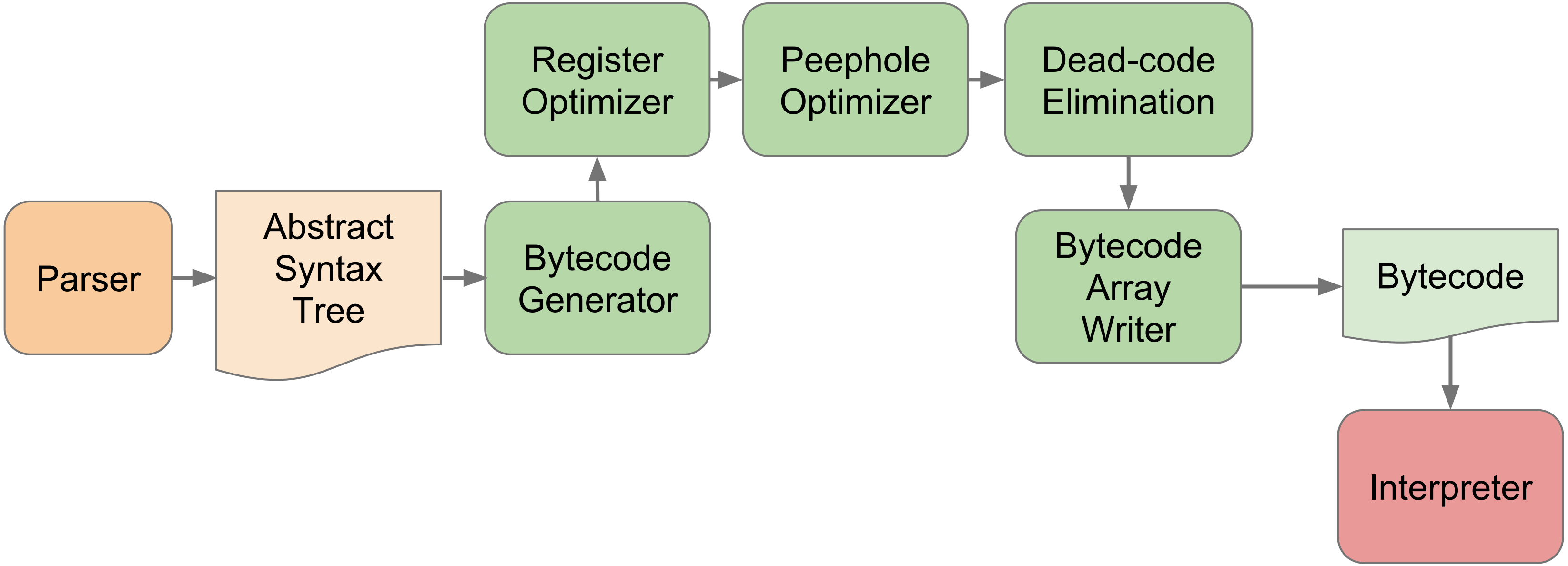
Ignition Bytecode Pipeline



Ignition Bytecode Pipeline



Ignition Bytecode Pipeline



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

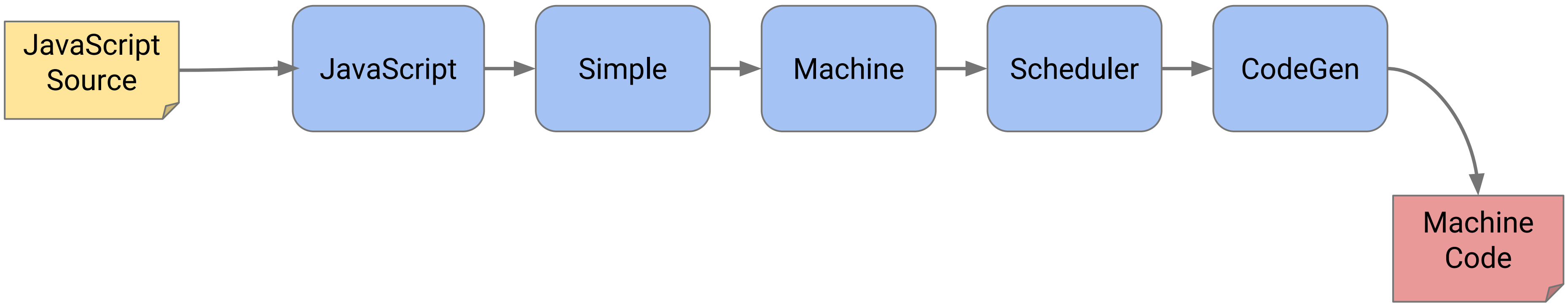
Building the Ignition Interpreter

- ✗ 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

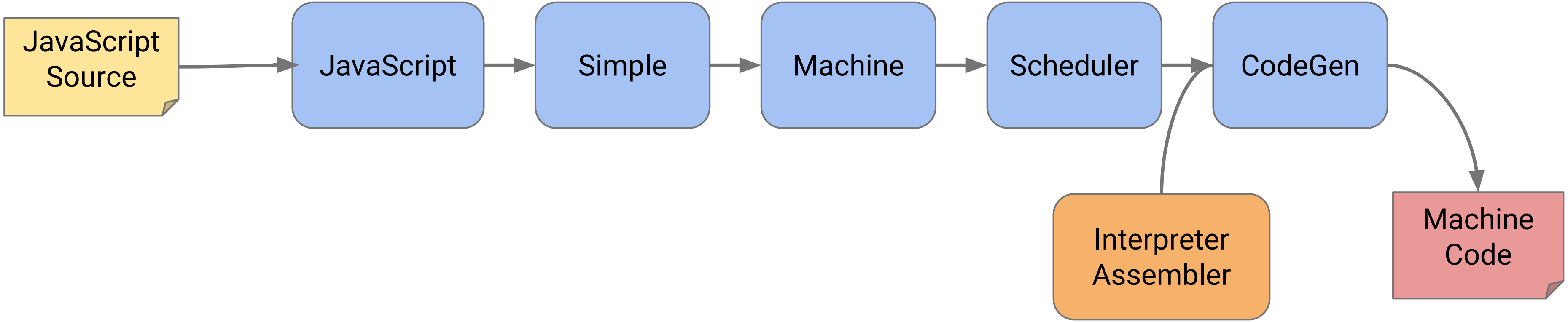
- ✗ Write in C++
  - Need trampolines between Interpreted and JITed functions
  - Can't interoperate with fast code-stubs
- ✗ Hand-crafted assembly code
  - Would need to be ported to 9 architectures
- ✔ Backend of the TurboFan Compiler
  - Write-once in macro-assembly
  - Architecture specific instruction selection optimizations for free
  - Relatively painless interoperability with existing code-stubs

TurboFan Pipeline

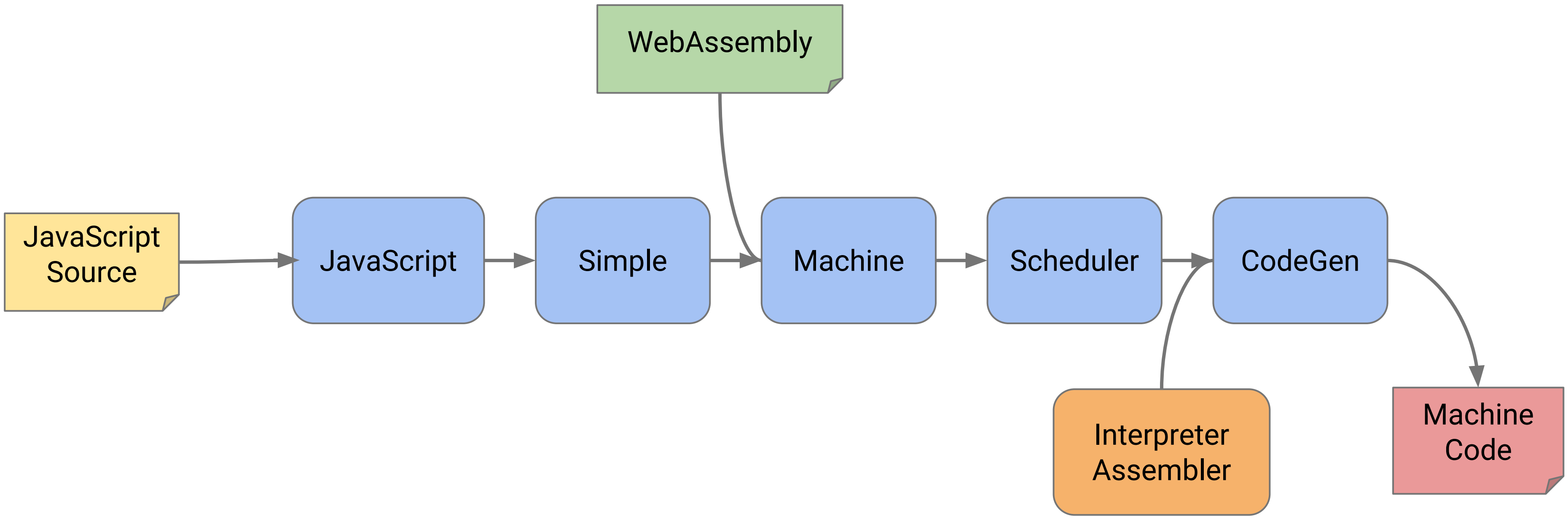


Google

TurboFan Pipeline



TurboFan Pipeline



Building an Interpreter using TurboFan

```
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();
}
```

Building an Interpreter using TurboFan

```
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

```
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

```
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

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

Indirect Threaded Bytecode Dispatch

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

Ldar

```
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
```

Indirect Threaded Bytecode Dispatch

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

Ldar

```
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

Indirect Threaded Bytecode Dispatch

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

Ldar

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

Ldar

```
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

Advance to next bytecode

Indirect Threaded Bytecode Dispatch

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

Ldar

```
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
Advance to next bytecode
Load next bytecode

Indirect Threaded Bytecode Dispatch

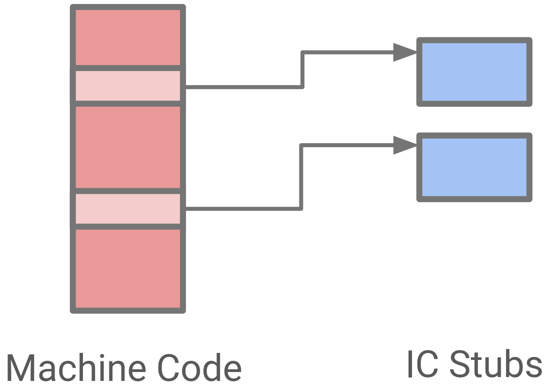
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Ldar

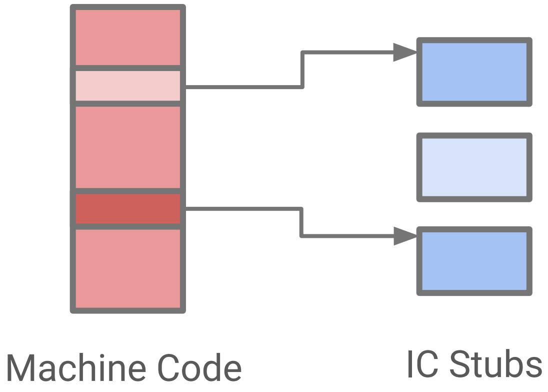
```
movsxbq rax,[r14+r12*1+0x1]
movq rax,[rbp+rax*8]
leaq r12,[r12+0x2]
movzxbl rbx,[r12+r14*1]
movq rbx,[r15+rbx*8]
jmp rbx
```

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

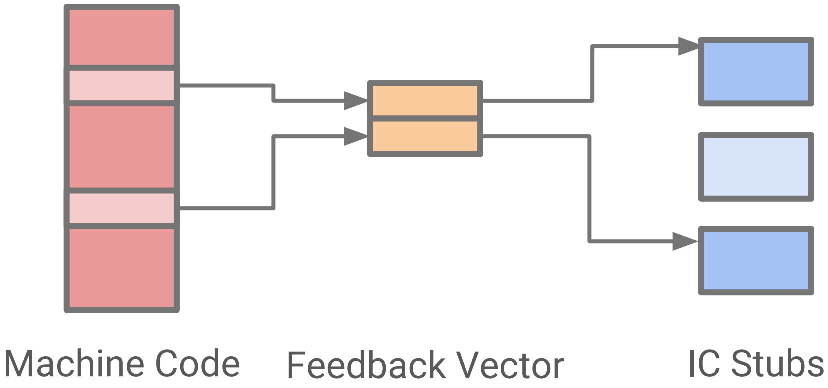
Inline Caches with Code Patching



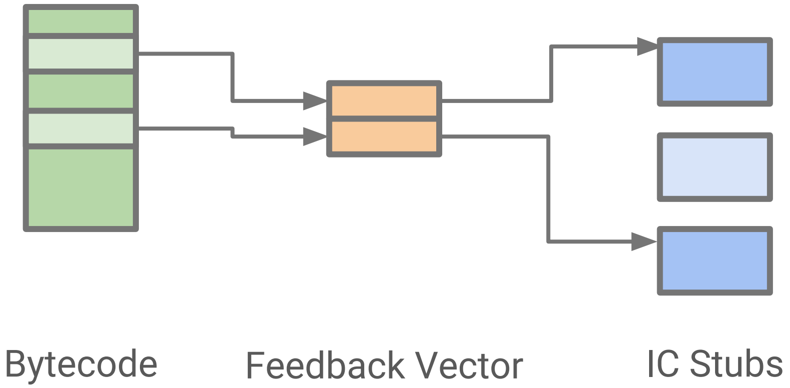
Inline Caches with Code Patching



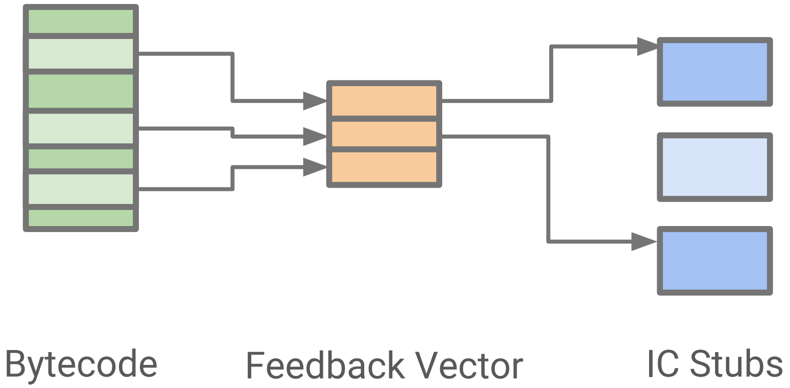
Inline Caches with Type Feedback Vector



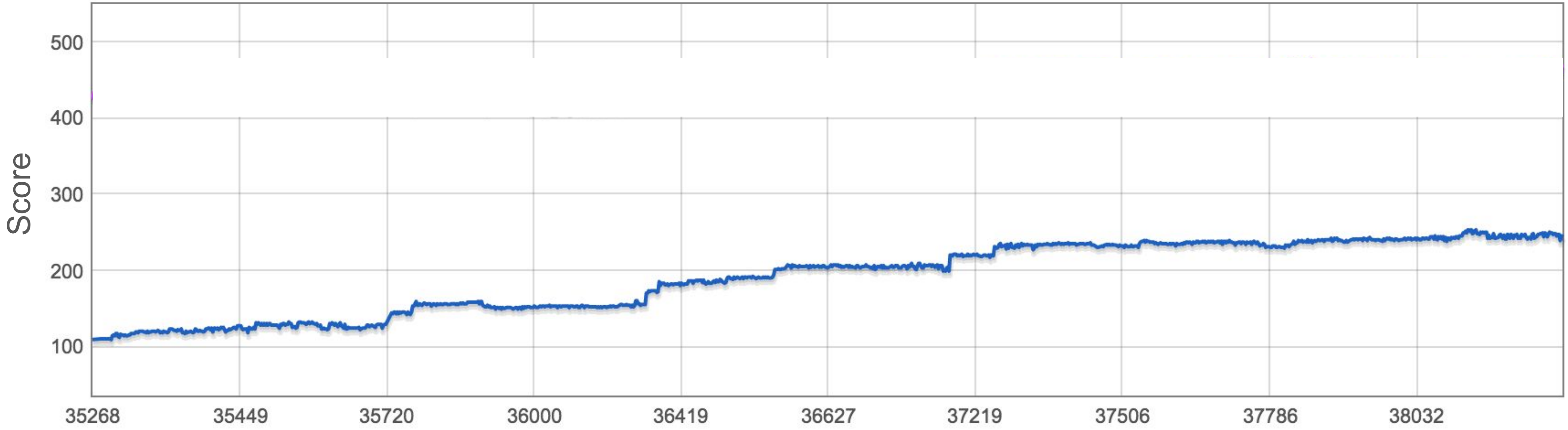
Inline Caches with Type Feedback Vector



Inline Caches with Type Feedback Vector



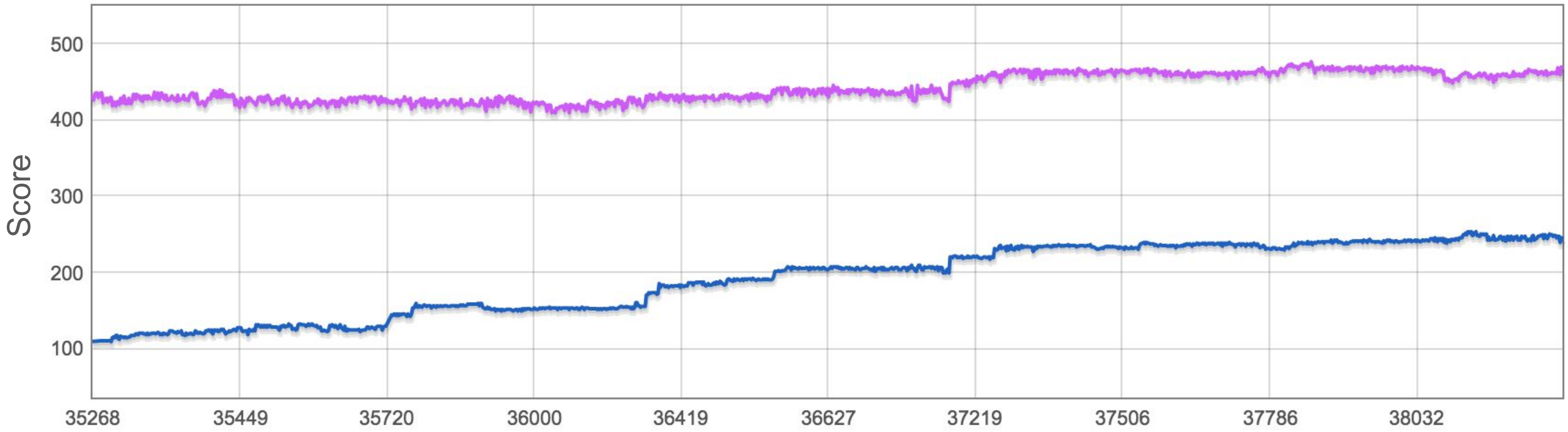
Ignition vs Full-Codegen



Octane (Nexus 5)

Crankshaft and TurboFan disabled

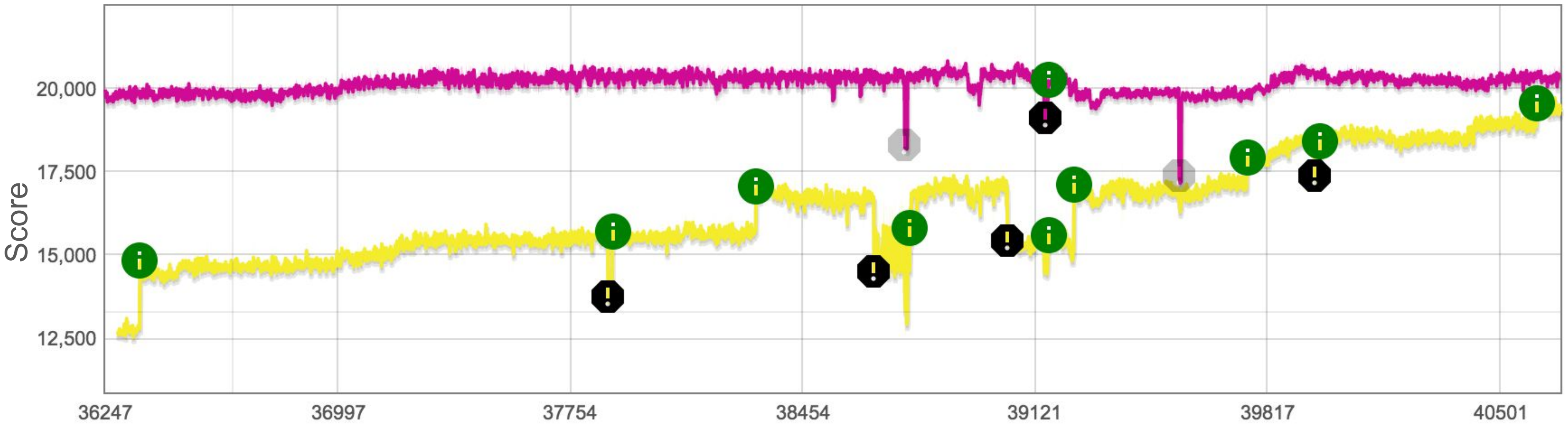
Ignition vs Full-Codegen



Octane (Nexus 5)

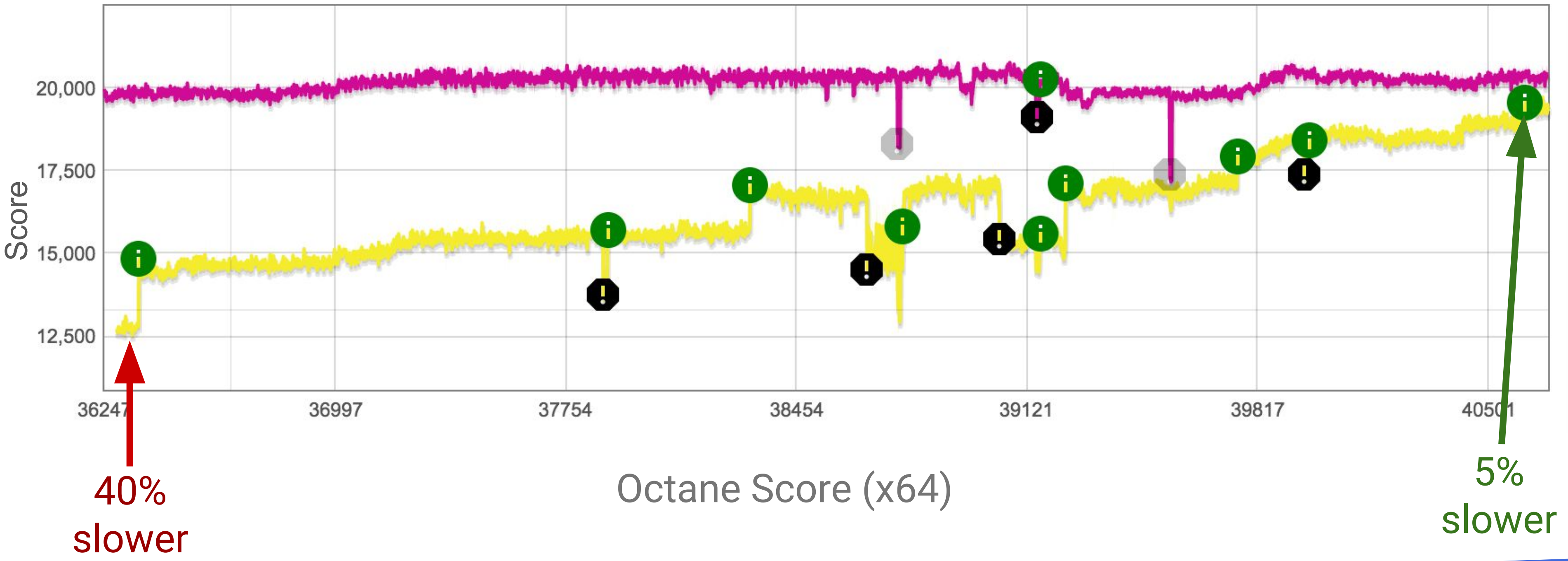
Crankshaft and TurboFan disabled

Ignition vs Default

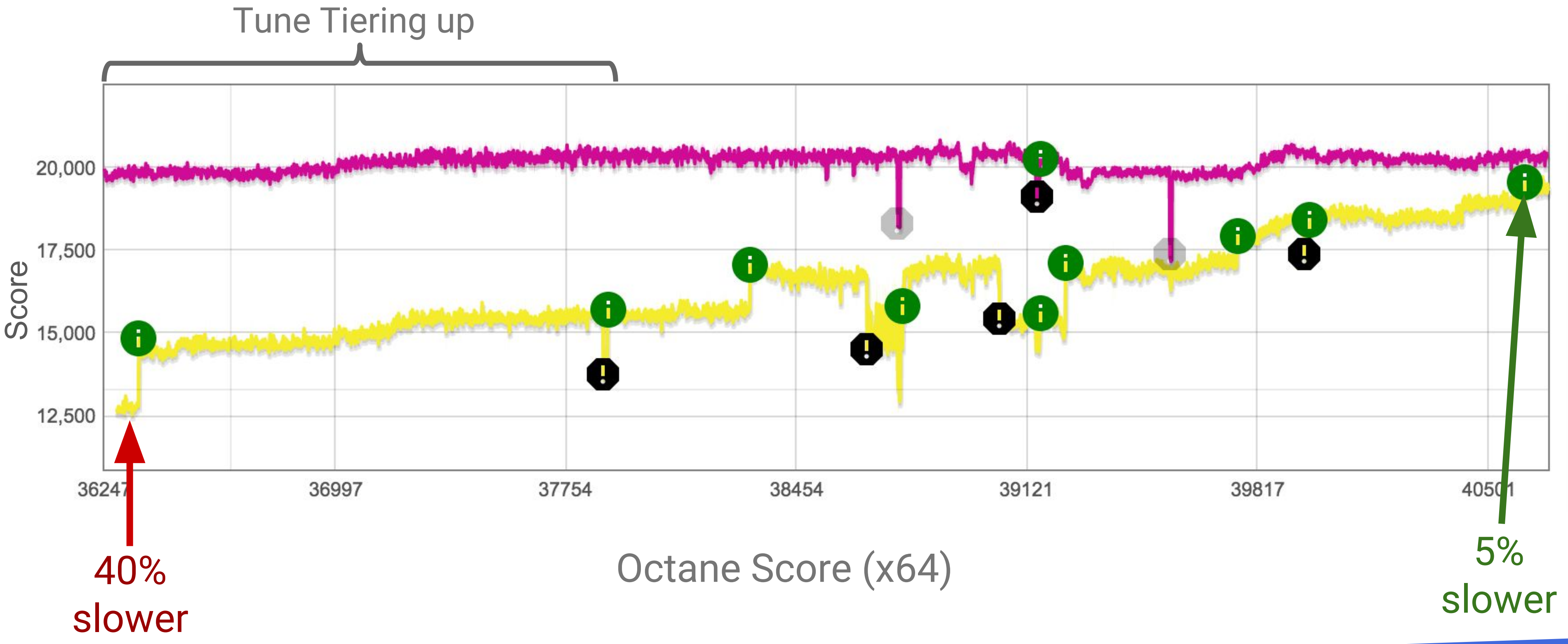


Octane Score (x64)

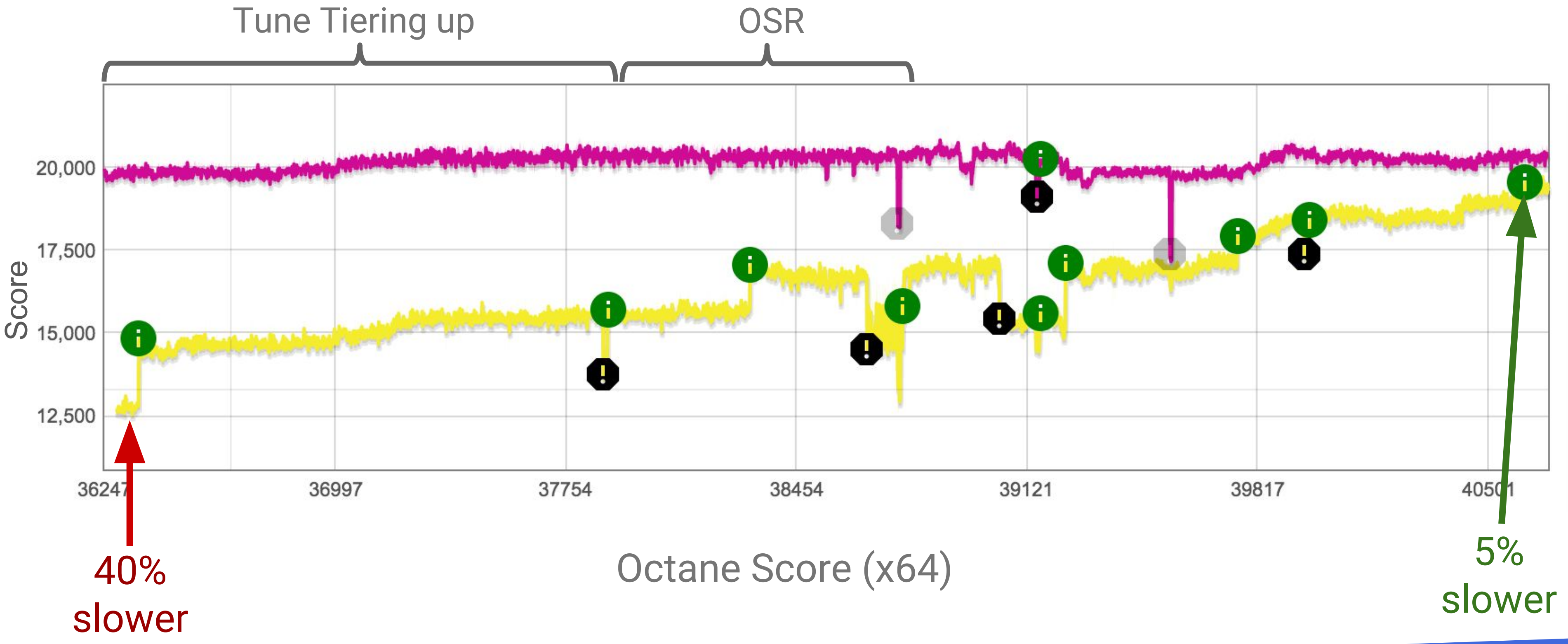
Ignition vs Default



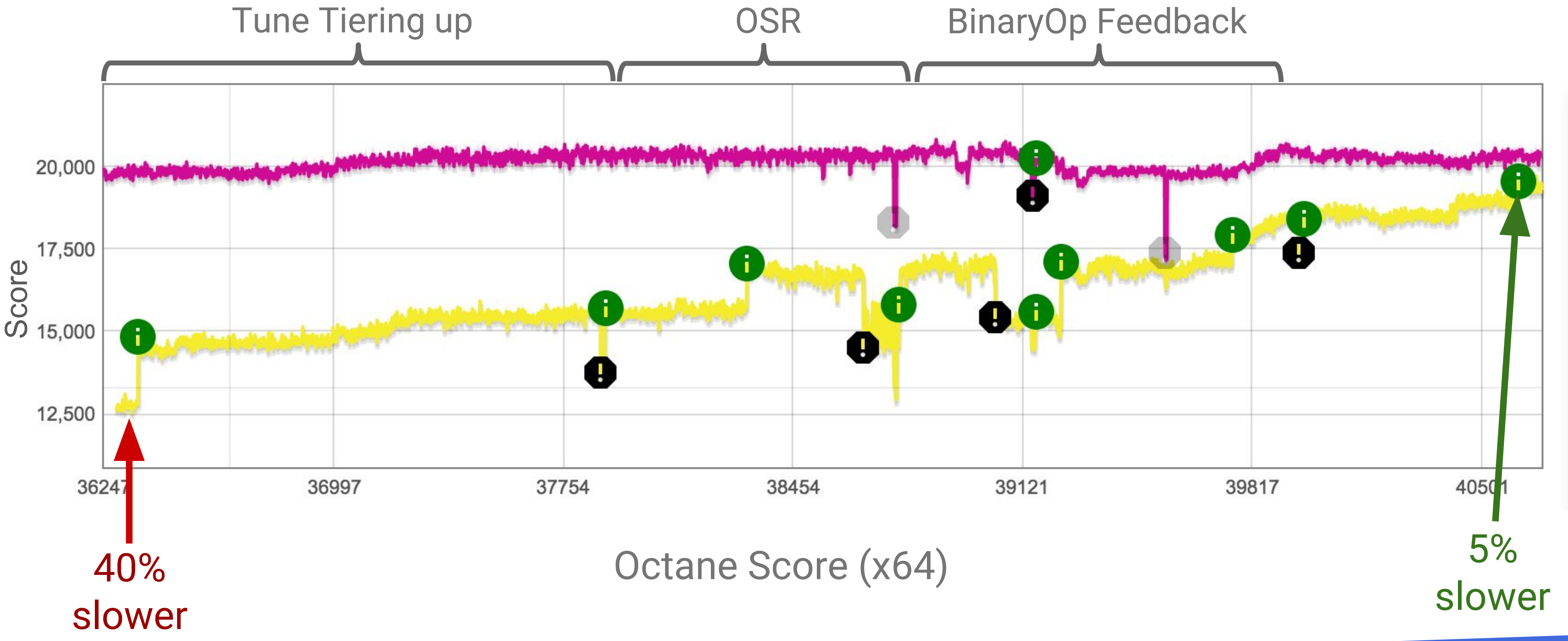
Ignition vs Default



Ignition vs Default

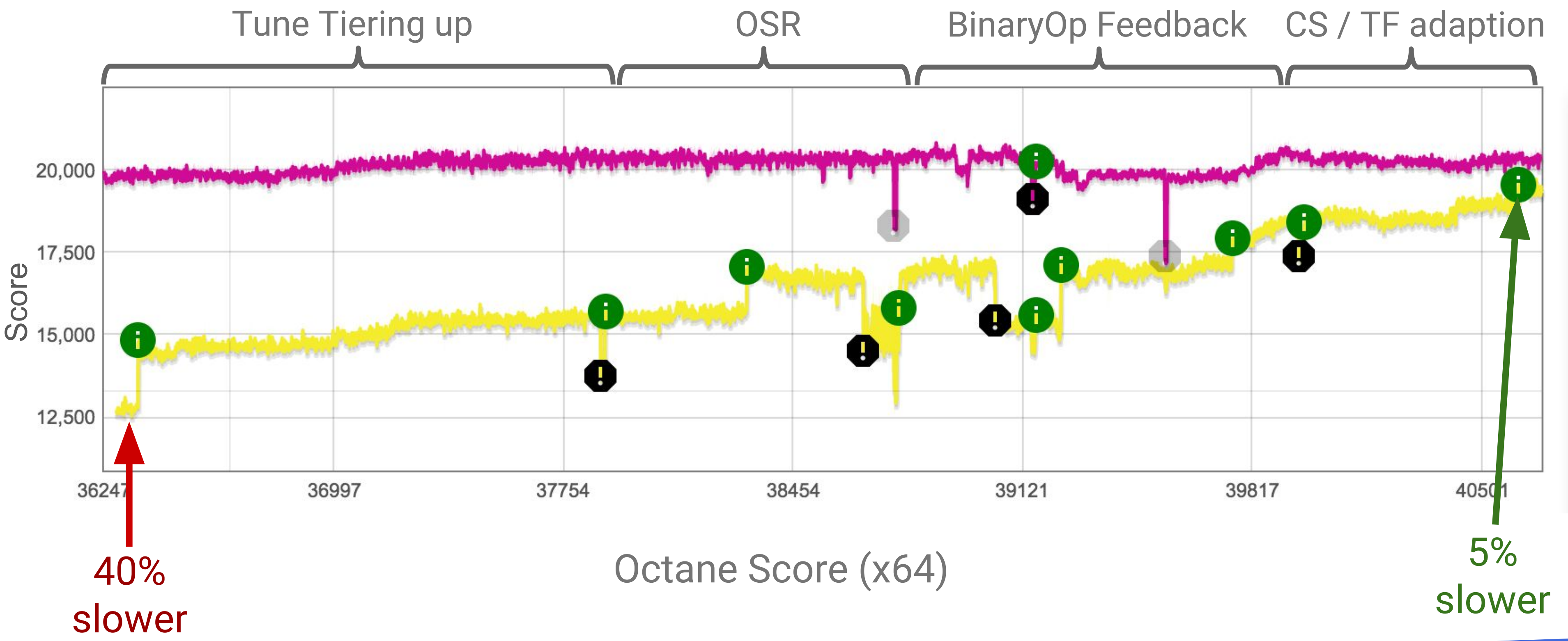


Ignition vs Default



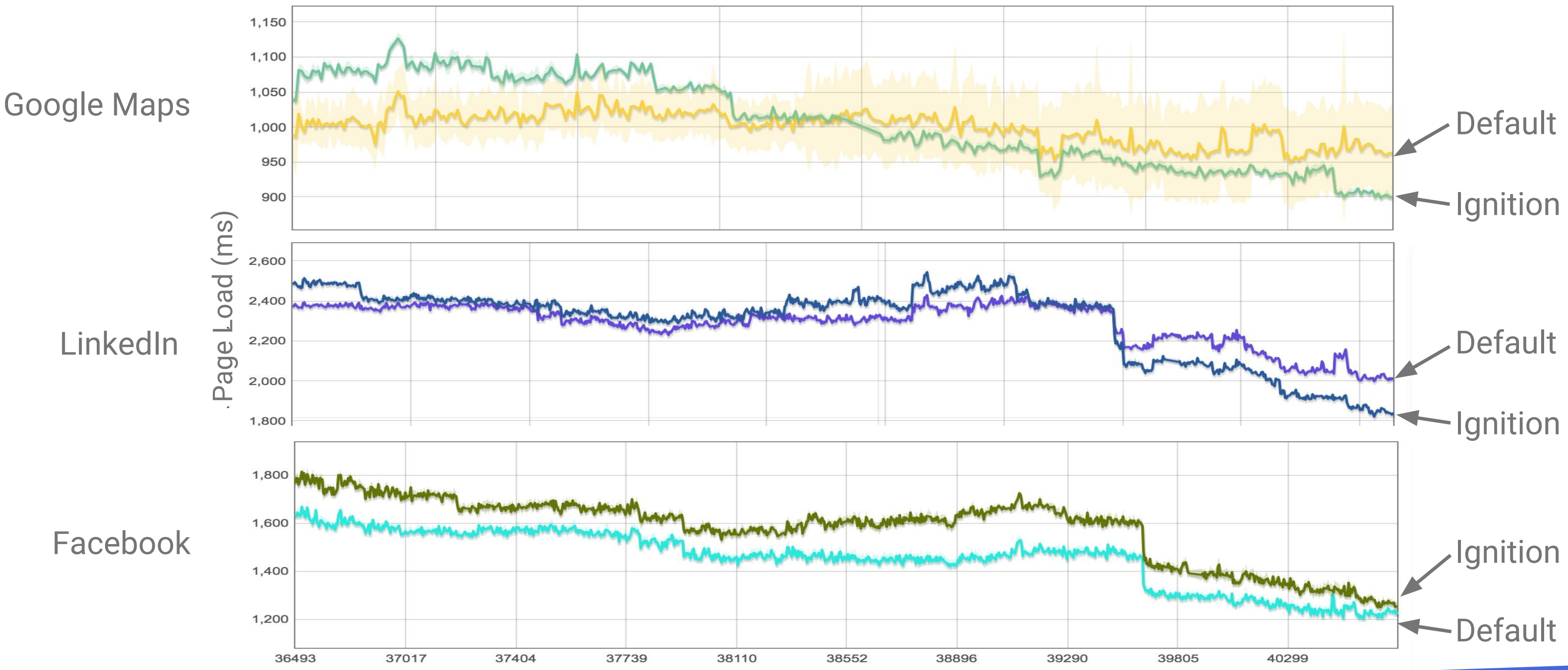
Google

Ignition vs Default

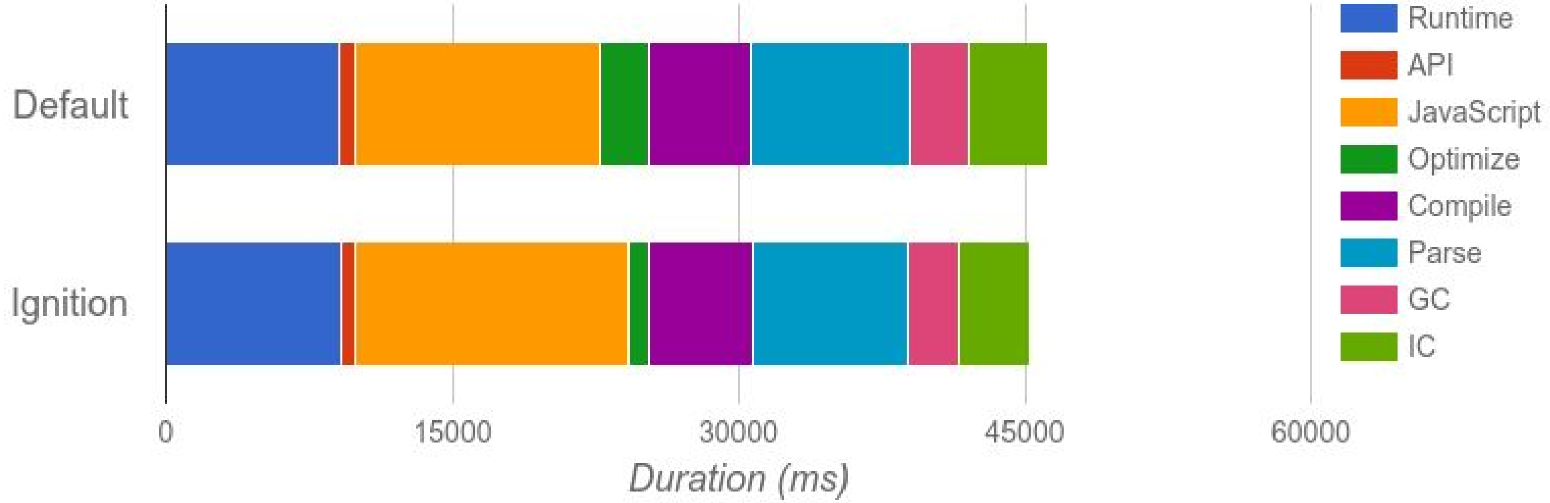


Google

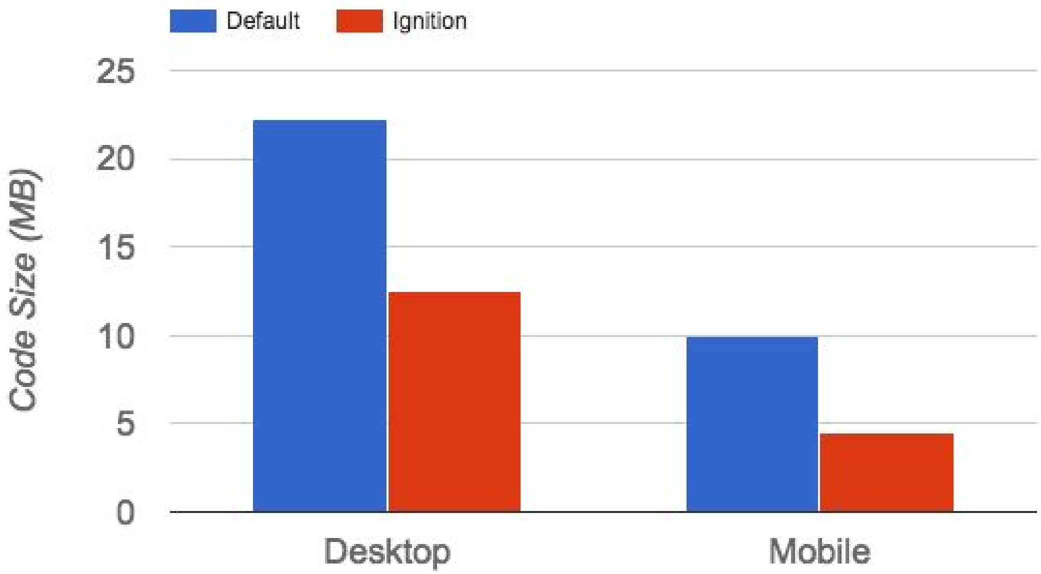
Real Websites



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

Non-Local Flow Control
Throw
ReThrow
Return

Literals
CreateRegExpLiteral
CreateArrayLiteral
CreateObjectLiteral

Load Property Operations
LdaNamedProperty
LdaKeyedProperty
KeyedLoadICStrict

Store Property Operations
StoreICSloppy
StoreICStrict
KeyedStoreICSloppy
KeyedStoreICStrict

Complex Flow Control
ForInPrepare
ForInNext
ForInDone
ForInStep

Generators
SuspendGenerator
ResumeGenerator