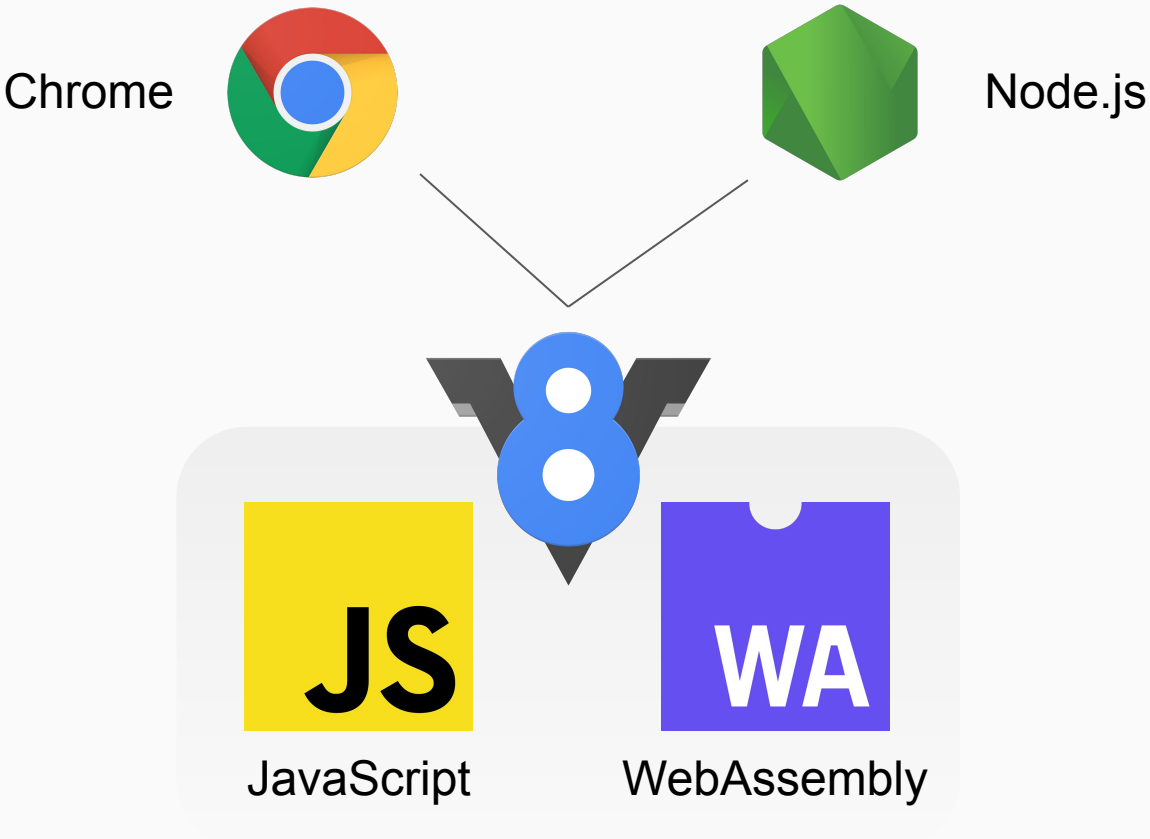
V8 Torque

Building a Type-Safeish DSL
to Implement JavaScript

Tobias Tebbi
V8 Team, Google

JFokus VM Tech Summit 2020


Chrome
Node.js
JS
WA
JavaScript
WebAssembly

There’s a Programming Language Hiding in V8...

V8 Torque

- statically typed
- garbage collected
- compatible with JavaScript
- only used inside of V8, at the moment ~20K lines of code

```
125   // array is not large enough, create and return a new, l
126   // contains all previously elements and the new element.
127   macro StoreAndGrowFixedArray<T: type>(
128       fixedArray: FixedArray, index: intptr, element: T):
129     const length: intptr = fixedArray.length_intptr;
130     assert(index <= length);
131     if (index < length) {
132       fixedArray.objects[index] = element;
133       return fixedArray;
134     } else
135       deferred {
136         const newLength: intptr = CalculateNewElementsCapa
137         assert(index < newLength);
138         const newfixedArray: FixedArray =
139             ExtractFixedArray(fixedArray, 0, length, newLe
140         newfixedArray.objects[index] = element;
141         return newfixedArray;
142       }
143   }
144
145   // Contains the information necessary to create a single
146   // flattened one or two byte string.
147   // The buffer is maintained and updated by Buffer constr
```

Why?

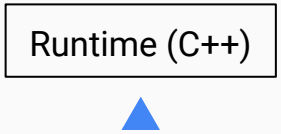
How to implement the JavaScript standard library?

Example
Array.prototype.indexOf

22.1.3.12Array.prototype.indexOf (searchElement [, fromIndex])
When the **`indexOf`** method is called with one or two arguments, the following steps are taken:

1. Let O be ? ToObject(**this** value).
2. Let len be ? ToLength(? Get(O, **`"length"`**)).
3. If len is 0, return -1.
4. Let n be ? ToInteger(fromIndex).
5. Assert: If fromIndex is **undefined**, then n is 0.
6. If n ≥ len, return -1.
7. If n ≥ 0, then
    a. If n is **-0**, let k be **+0**; else let k be n.
8. Else n < 0,
    a. Let k be len + n.
    b. If k < 0, set k to 0.
9. Repeat, while k < len
    a. Let kPresent be ? HasProperty(O, ! ToString(k)).
    b. If kPresent is **true**, then
        i. Let elementK be ? Get(O, ! ToString(k)).
        ii. Let same be the result of performing Strict Equality Comparison searchElement === elementK.
        iii. If same is **true**, return k.
    c. Increase k by 1.
10. Return -1.

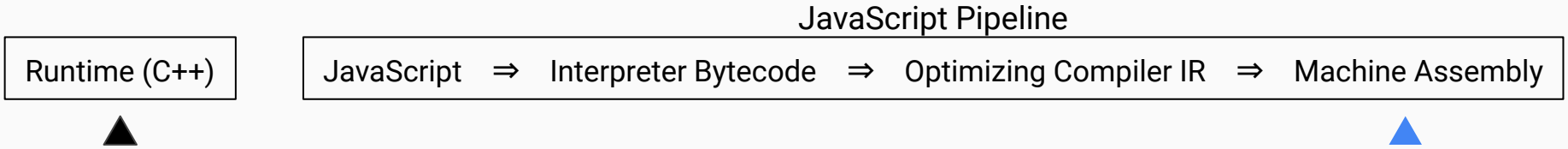
Implementation Options in V8



Use C++ API

+ High-level language
+ Compiler optimizations
- Expensive transition to our JavaScript ABI
- Handling GC pointers is expensive.

Implementation Options in V8



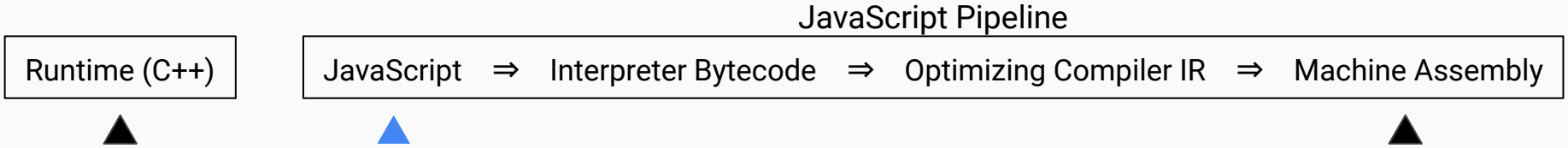
Use C++ API

+ High-level language
+ Compiler optimizations
- Expensive transition to our JavaScript ABI
- Handling GC pointers is expensive.

Use Hand-Written Assembly

+ Full Control
- Low productivity
- Error-prone
- We support 7 platforms: doesn't scale.

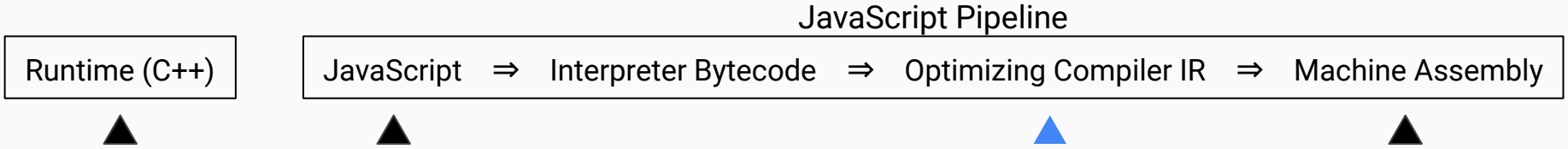
Implementation Options in V8



Use JavaScript

+ Portable, Readable
+ ABI and GC compatible
- Needs warm-up to be optimized
- JS semantics easily leaks (monkey-patching …)
- Exposing VM internals is dangerous

Implementation Options in V8



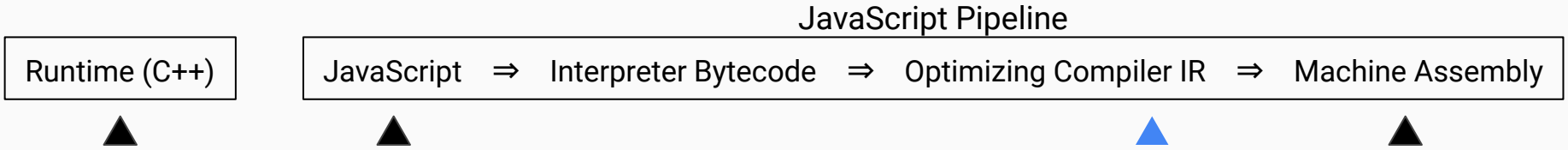
Use JavaScript

+ Portable, Readable
+ ABI and GC compatible
- Needs warm-up to be optimized
- JS semantics easily leaks (monkey-patching …)
- Exposing VM internals is dangerous

Use our optimizing compiler

+ Portable
+ ABI and GC compatible
+ Low-level control
+ Internals Exposable
+ Full control of semantics
- Low-level “language”

Implementation Options in V8

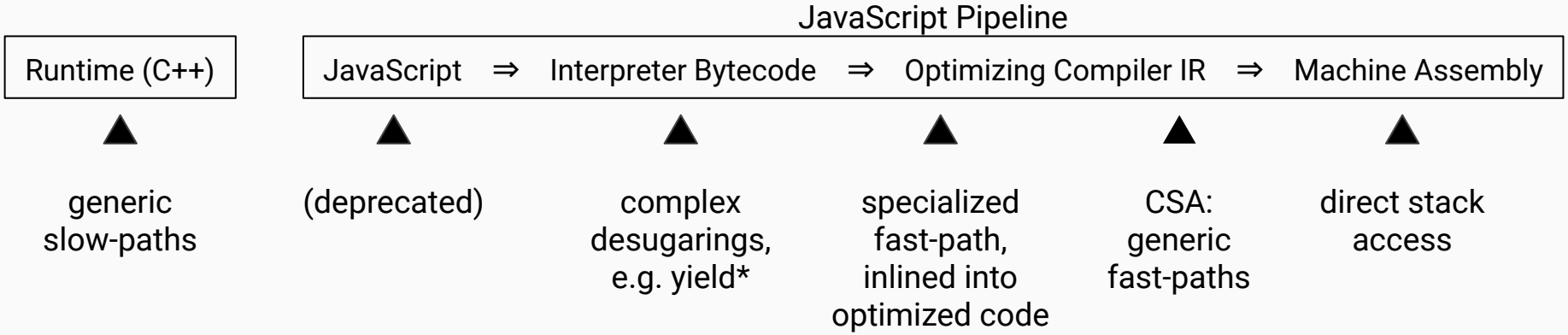


The CodeStubAssembler (CSA)

Use our optimizing compiler

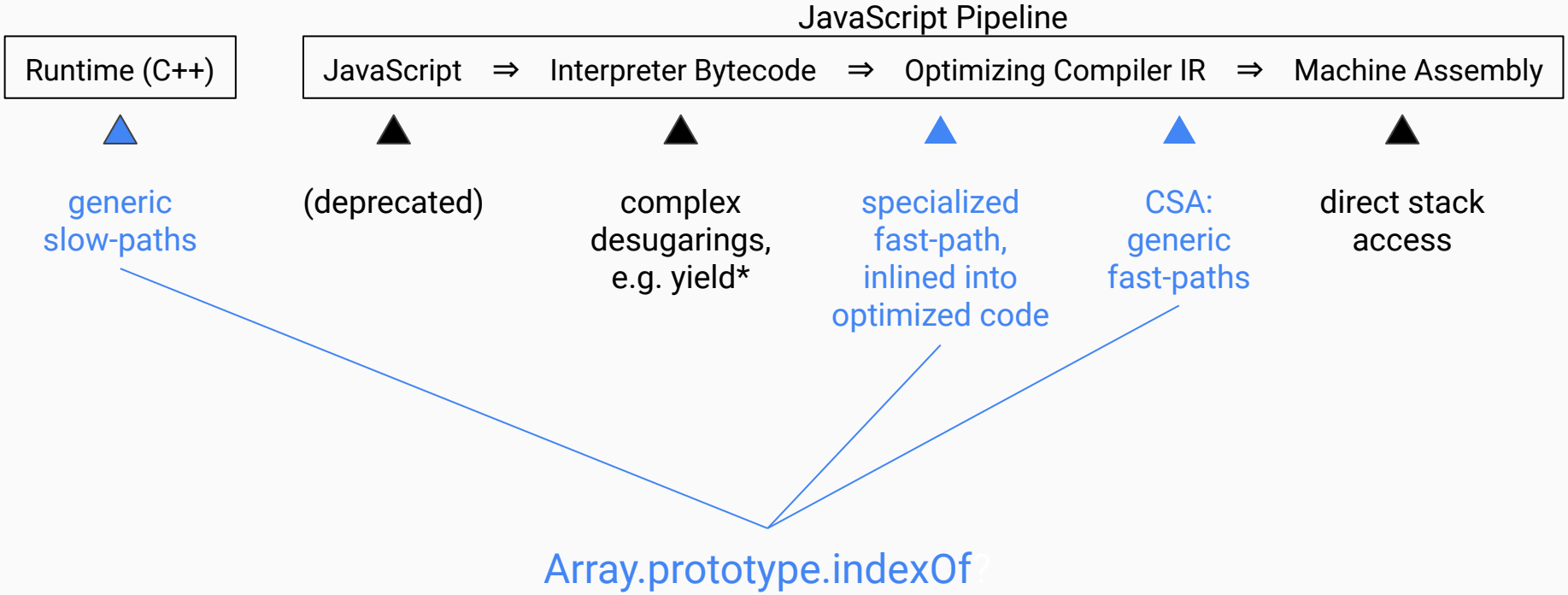
- Based on the back-end of Turbofan, our optimizing JS compiler.
- Just before register allocation and code generation:
  a platform-independent assembler.
- We wrote a lot of code in CSA (~50K lines):
  builtins, interpreter, inline caches

We do all of it!



Array.prototype.indexOf?

We do all of it!



CSA: the CodeStubAssembler

```
TF_BUILTIN(ArrayOf, ArrayPopulatorAssembler) {
 TNode<Int32T> argc =
   UncheckedCast<Int32T>(Parameter(
                         Descriptor::kJSActualArgumentsCount));
 TNode<Smi> length = SmiFromInt32(argc);
 TNode<Context> context = CAST(Parameter(Descriptor::kContext));
 CodeStubArguments args(this, length, nullptr,
                        ParameterMode::SMI_PARAMETERS);
 TNode<Object> receiver = args.GetReceiver();

 TVARIABLE(Object, array);
 Label is_constructor(this), is_not_constructor(this),
       fill_array(this);
 GotoIf(TaggedIsSmi(receiver), &is_not_constructor);
 Branch(IsConstructor(CAST(receiver)), &is_constructor,
                                       &is_not_constructor);

 BIND(&is_constructor);
 array = Construct(context, CAST(receiver), length);
 Goto(&fill_array);

 BIND(&is_not_constructor);
 array = ArrayCreate(context, length);
 Goto(&fill_array);

 BIND(&fill_array);
 BuildFastLoop(SmiConstant(0), length,
               [&](Node* index) {
                 CallRuntime(
                     Runtime::kCreateDataProperty, context,
                     static_cast<Node*>(array.value()),
                     index, args.AtIndex(index,
                             ParameterMode::SMI_PARAMETERS));
               },
               1, ParameterMode::SMI_PARAMETERS,
IndexAdvanceMode::kPost);

 GenerateSetLength(context, array.value(), length);
 args.PopAndReturn(array.value());
}
```

CSA: the CodeStubAssembler

```
TF_BUILTIN(ArrayOf, ArrayPopulatorAssembler) {
 TNode<Int32T> argc =
   UncheckedCast<Int32T>(Parameter(
                         Descriptor::kJSActualArgumentsCount));
 TNode<Smi> length = SmiFromInt32(argc);
 TNode<Context> context = CAST(Parameter(Descriptor::kContext));
 CodeStubArguments args(this, length, nullptr,
                        ParameterMode::SMI_PARAMETERS);
 TNode<Object> receiver = args.GetReceiver();
```

```
TVARIABLE(Object, array);
Label is_constructor(this), is_not_constructor(this),
      fill_array(this);
GotoIf(TaggedIsSmi(receiver), &is_not_constructor);
Branch(IsConstructor(CAST(receiver)), &is_constructor,
                                      &is_not_constructor);

BIND(&is_constructor);
array = Construct(context, CAST(receiver), length);
Goto(&fill_array);
```

if-then-else

```
BIND(&is_not_constructor);
array = ArrayCreate(context, length);
Goto(&fill_array);
```

a for-loop

```
BIND(&fill_array);
BuildFastLoop(SmiConstant(0), length,
              [&](Node* index) {
                CallRuntime(
                    Runtime::kCreateDataProperty, context,
                    static_cast<Node*>(array.value()),
                    index, args.AtIndex(index,
                            ParameterMode::SMI_PARAMETERS));
              },
              1, ParameterMode::SMI_PARAMETERS,
IndexAdvanceMode::kPost);
```

```
 GenerateSetLength(context, array.value(), length);
 args.PopAndReturn(array.value());
}
```

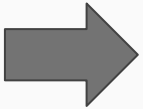
The same with Torque ...

- Structured Control Flow
- Operators
- Expressive Type System
- Auto-Generated Boilerplate

```
ArrayOf(js-implicit context: NativeContext, receiver: JSAny)
       (...arguments): JSAny {
  let a: JSReceiver;
  typeswitch (receiver) {
    case (c: Constructor): {
      a = Construct(c, Convert<Smi>(arguments.length));
    }
    case (JSAny): {
      a = ArrayCreate(Convert<Smi>(arguments.length));
    }
  }
  for (let k: intptr = 0; k < arguments.length; k++) {
    let kValue: Object = arguments[k];
    CreateDataProperty(a, Convert<Smi>(k), kValue);
  }
  SetProperty(a, kLengthString, len);
  return a;
}
```

What is Torque?

CSA semantics
+
JS-like syntax
+
static types
+
compiler optimizations
+
safe abstractions



- Development Velocity
- Security
- Better Tooling

Design Principles

- Prioritize safe and maintainable code.
- (Approximating) Memory-Safety
- Explicit about invariants, implicit about boilerplate.
- A new way to respond to systematic exploit classes: improve Torque’s static analysis.

Discover the language that’s hiding in V8.

A Language from Scratch

Pragmatic initial design:

- Compile to CSA C++ code.
- Typescript-like Syntax
- No Optimizations
- Zero-Cost Abstractions

```
// A type system of abstract types, mapping to the CSA types
type Tagged generates 'TNode<Object>';
type Smi extends Tagged generates 'TNode<Smi>';
type HeapObject extends Tagged generates 'TNode<HeapObject>';
```

garbage-collected pointer

A Language from Scratch

Pragmatic initial design:

- Compile to CSA C++ code.
- Typescript-like Syntax
- No Optimizations
- Zero-Cost Abstractions

```
// A type system of abstract types, mapping to the CSA types
type Tagged generates 'TNode<Object>';
type Smi extends Tagged generates 'TNode<Smi>';
type HeapObject extends Tagged generates 'TNode<HeapObject>';

// Import existing CSA operations.
extern operator '+' macro SmiAdd(Smi, Smi): Smi;
```

```
TNode<Smi> CodeStubAssembler::SmiAdd(TNode<Smi> a, TNode<Smi> b);
```

A Language from Scratch

Pragmatic initial design:

- Compile to CSA C++ code.
- Typescript-like Syntax
- No Optimizations
- Zero-Cost Abstractions

```
// A type system of abstract types, mapping to the CSA types
type Tagged generates 'TNode<Object>';
type Smi extends Tagged generates 'TNode<Smi>';
type HeapObject extends Tagged generates 'TNode<HeapObject>';

// Import existing CSA operations.
extern operator '+' macro SmiAdd(Smi, Smi): Smi;
```

```
TNode<Smi> CodeStubAssembler::SmiAdd(TNode<Smi> a, TNode<Smi> b);
```

```
// Define builtins with Javascript linkage ...
javascript builtin ExampleFunction(js-implicit receiver: Object)
                                   (...arguments): Object {
  return 1 + Cast<Smi>(receiver) otherwise return 0;
}
```

Macros

Always inlined

Inherited from CSA

Zero-Cost ... but lead to code explosion.

```
macro Square(x: Smi): Smi {
  return x * x;
}

builtin FourthPower(x: Smi): Smi {
  // This emits two multiplication instructions.
  return Square(Square(x));
}
```

Labels: Goto, but structured.

Turbofan only supports structured control flow.

CSA's goto was useful, but unsafe.

```
for (let i: Smi = 0; i < length; ++i) {
  if (!IsSafe(input[i])) goto Bailout;
}
// Fast-path ...

label Bailout
return CallRuntime(...);
```

Labels: Goto, but structured.

Turbofan only supports structured control flow.

CSA's goto was useful, but unsafe.

Exception semantics: safe, familiar

```
try {
  for (let i: Smi = 0; i < length; ++i) {
    if (!IsSafe(input[i])) goto Bailout;
  }
  // Fast-path ...

} label Bailout {
  return CallRuntime(...);
}
```

Labels: Goto, but structured.

Turbofan only supports structured control flow.

CSA's goto was useful, but unsafe.

Exception semantics: safe, familiar

Supported across macros.

```
try {
  for (let i: Smi = 0; i < length; ++i) {
    if (!IsSafe(input[i])) goto Bailout;
  }
  // Fast-path ...

} label Bailout {
  return CallRuntime(...);
}

macro TryFastPath(x: Object) label Bailout {
  let x = Cast<Smi>(x) otherwise goto Bailout;
  // Continue on fast path ...
}
```

Labels: Goto, but structured.

Turbofan only supports structured control flow.

CSA's goto was useful, but unsafe.

Exception semantics: safe, familiar

Supported across macros.

Can have arguments: still zero-cost

```
try {
  for (let i: Smi = 0; i < length; ++i) {
    if (!IsSafe(input[i])) goto Bailout(i);
  }
  // Fast-path ...

} label Bailout(i: Smi) {
  return CallRuntime(...);
}

macro TryFastPath(x: Object) label Bailout {
  let x = Cast<Smi>(x) otherwise goto Bailout;
  // Continue on fast path ...
}
```

Union Types

Safe runtime distinction of types.

```
type Number = Smi | HeapNumber;
```

Union Types

Safe runtime distinction of types.

Typeswitch: Guarantees type coverage.

```
type Number = Smi | HeapNumber;

macro Plus1(x: Number) : Number {
  typeswitch(x) {
    case (x: Smi): {
      return x + 1;
    }
    case (x: HeapNumber): {
      return new HeapNumber{x.value + 1};
    }
  }
}
```

Structs

Zero-Cost:
Always desugared to SSA values

Value Semantics

```
struct KeyValuePair {
  key: Object;
  value: Object;
}
```

Structs

Zero-Cost:
Always desugared to SSA values

Value Semantics

```
struct KeyValuePair {
  key: Object;
  value: Object;
}

let pair: KeyValuePair = KeyValuePair{Null, Undefined};
pair.value = True;
```

Defining Heap Object Layouts

```
class HeapNumber extends HeapObject {
  value: float64;
}
```

Defining Heap Object Layouts

```
class HeapNumber extends HeapObject {
  value: float64;
}

macro AllocateHeapNumber(value: float64) {
  return new HeapNumber{map: kHeapNumberMap, value: value};
}
```

Defining Heap Object Layouts

```
class HeapNumber extends HeapObject {
  value: float64;
}

macro AllocateHeapNumber(value: float64) {
  return new HeapNumber{map: kHeapNumberMap, value: value};
}

class SeqOneByteString extends HeapObject {
  hash_field: uint32;
  length: int32;
  chars[length]: char8;
}
```

Defining Heap Object Layouts

```
class HeapNumber extends HeapObject {
  value: float64;
}

macro AllocateHeapNumber(value: float64) {
  return new HeapNumber{map: kHeapNumberMap, value: value};
}

class SeqOneByteString extends HeapObject {
  hash_field: uint32;
  length: int32;
  chars[length]: char8;
}

extern class SmallOrderedHashSet extends HeapObject {
  number_of_elements: uint8;
  number_of_deleted_elements: uint8;
  number_of_buckets: uint8;
  data_table[number_of_buckets * 2]: JSAny|TheHole;
  hash_table[number_of_buckets]: uint8;
  chain_table[number_of_buckets * 2]: uint8;
}
```

Defining Heap Object Layouts

Torque generates:

Heap Object Layouts ✔

CSA/C++ Field Accessors ✔

Object Printers (✔)

GC Visitors (WIP)

```
class HeapNumber extends HeapObject {
  value: float64;
}

macro AllocateHeapNumber(value: float64) {
  return new HeapNumber{map: kHeapNumberMap, value: value};
}

class SeqOneByteString extends HeapObject {
  hash_field: uint32;
  length: int32;
  chars[length]: char8;
}

extern class SmallOrderedHashSet extends HeapObject {
  number_of_elements: uint8;
  number_of_deleted_elements: uint8;
  number_of_buckets: uint8;
  data_table[number_of_buckets * 2]: JSAny|TheHole;
  hash_table[number_of_buckets]: uint8;
  chain_table[number_of_buckets * 2]: uint8;
}
```

Static Analysis

Example: Transient Types

V8 heap objects can change layout at runtime.

```
let array : FastJSArray = ...;

// Looks harmless, but what if x is
// {toString: function(){
//     array.length = 1000000000; } }
ToString(x);

// Oops, array is no longer fast ...
return array[5];
// ... heap corruption follows ...
```

Example: Transient Types

V8 heap objects can change layout at runtime.

Transient types encode this in the type system.

```
transient type FastJSArray extends JSArray
                           generates 'TNode<JSArray>';
extern transitioning macro ToString(Context, Object): String;

let array : FastJSArray = ...;

// Looks harmless, but what if x is
// {toString: function(){
//     array.length = 1000000000; } }
ToString(x);

// Oops, array is no longer fast...
return array[5];
// ... heap corruption follows ...
```

Example: Transient Types

V8 heap objects can change layout at runtime.

Transient types encode this in the type system.

```
transient type FastJSArray extends JSArray
                           generates 'TNode<JSArray>';
extern transitioning macro ToString(Context, Object): String;

let array : FastJSArray = ...;



//  ToString(x) invalidates FastJSArray
ToString(x);

// Compile Error: Accessing invalidated value
return array[5];
```

Building Abstractions

Witness of Speed

Frequent Pattern: Just check the map to validate a FastJSArray.

Transient types cannot encode that.

But they don’t need to.

```
const array : JSArray = fastArray;

ToString(x);

return UnsafeCast<FastJSArray>(array);
```

Witness of Speed

Frequent Pattern: Just check the map to validate a FastJSArray.

Transient types cannot encode that.

But they don’t need to.

```
const array : JSArray = fastArray;
const map: Map = array.map;

ToString(x);

if (array.map != map) goto Bailout;

return UnsafeCast<FastJSArray>(array);
```

Witness of Speed

Frequent Pattern: Just check the map to validate a FastJSArray.

Transient types cannot encode that.

But they don’t need to.

```
const array : JSArray = fastArray;
const map: Map = array.map;

ToString(x);

if (array.map != map) goto Bailout;

return UnsafeCast<FastJSArray>(array);
```

```
let array : FastJSArrayWitness = NewFastJSArrayWitness(fastArray);
ToString(x);
array.Recheck() otherwise Bailout;
return array.Get();
```

Witness of Speed

Frequent Pattern: Just check the map to validate a FastJSArray.

Transient types cannot encode that.

But they don’t need to.

```
struct FastJSArrayWitness {
  unstable: FastJSArray;
  stable: JSArray;
  map: Map;

  macro Get() { return this.unstable; }
  macro Recheck() labels CastError {
    if (this.stable.map != this.map) goto CastError;
    this.unstable = UnsafeCast<FastJSArray>(this.stable);
  }
}
macro NewFastJSArrayWitness(array: FastJSArray): FastJSArrayWitness {
  return FastJSArrayWitness{unstable: array, stable: array,
                            map: array.map};
}

let array : FastJSArrayWitness = NewFastJSArrayWitness(fastArray);
ToString(x);
array.Recheck() otherwise Bailout;
return array.Get();
```

Optimizations

Why an Optimizing Compiler?

Hand-Optimized code is hard to read and brittle.

It’s easy to introduce security bugs.

What about changing invariants?

Using Our Optimizing Compiler

We already have an optimizing compiler: Turbofan.

Rewire the CSA pipeline to use Turbofan optimization.

Current optimizations:

- Write-barrier elimination
- Load elimination
- Bounds check elimination

Tooling

Tooling

- Language Server + VS Code plugin
- DWARF source position information: break on Torque source line.
- Future: Full debugging support?

Future Directions

- We want it to be really hard to introduce accidental security bugs.
- Our interpreter is written in CSA, could be ported to Torque.
- Our runtime is implemented in a mixture of C++ and CSA. So far, C++ was the slow but convenient option. With Torque, this might change.
- The Torque/CSA pipeline only shares the back-end with optimized JavaScript. If we shared the complete pipeline, we could inline anything.