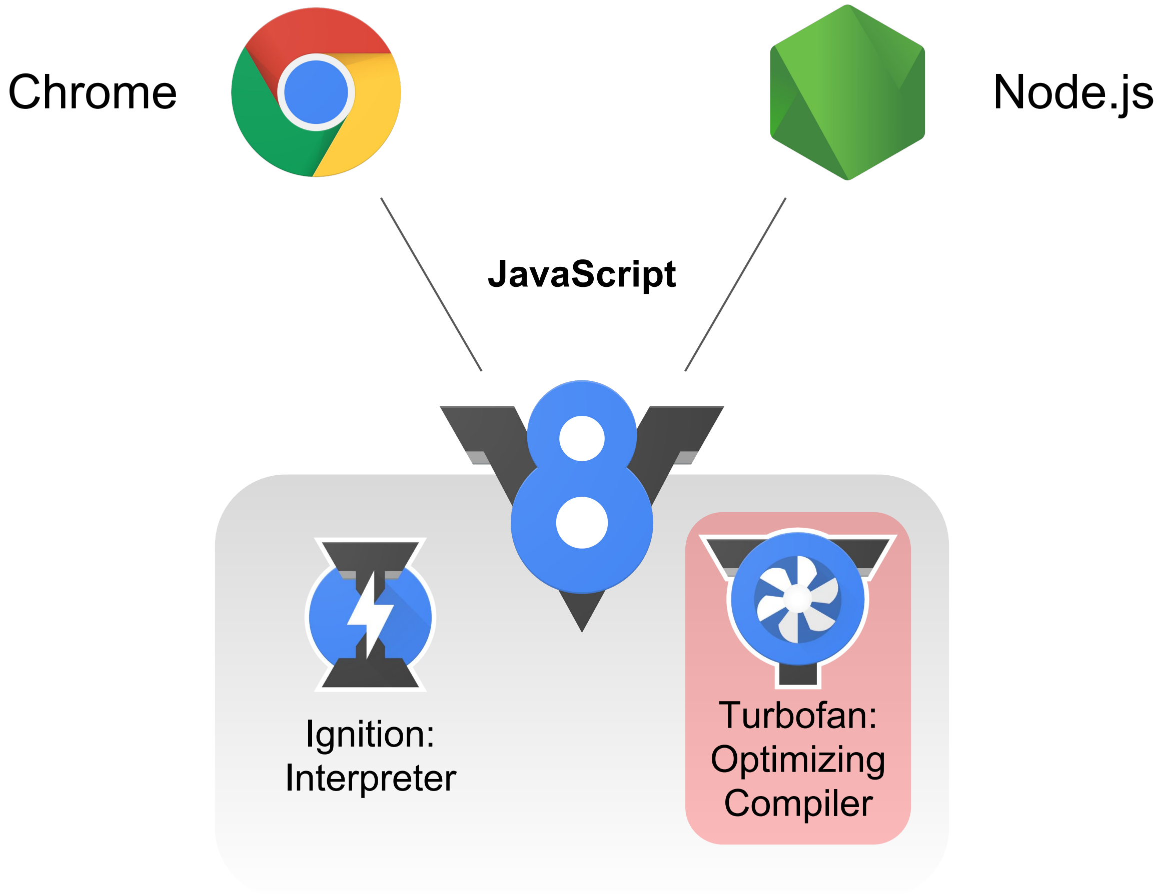
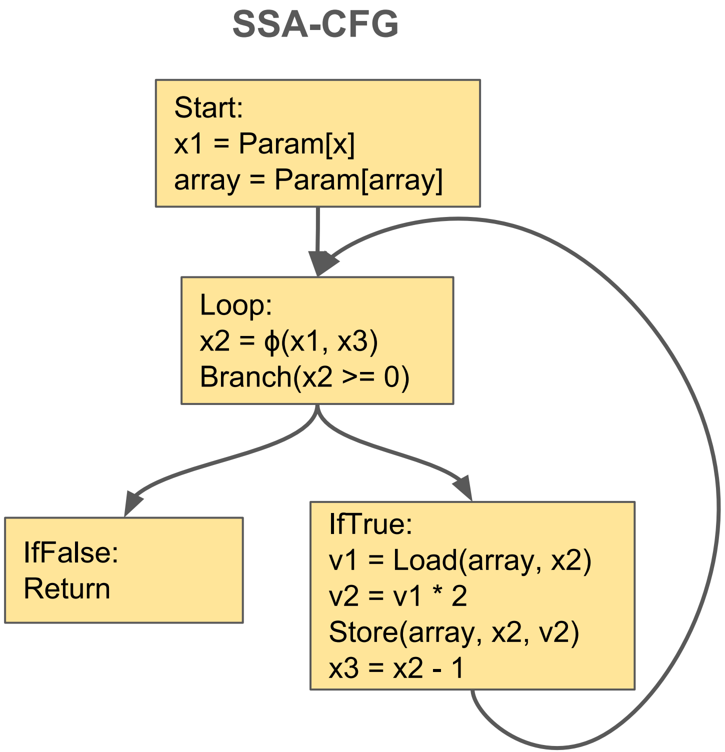
Escape Analysis in V8

Tobias Tebbi
V8 Team, Chrome, Google


Chrome
Node.js
JavaScript
Ignition:
Interpreter
Turbofan:
Optimizing
Compiler

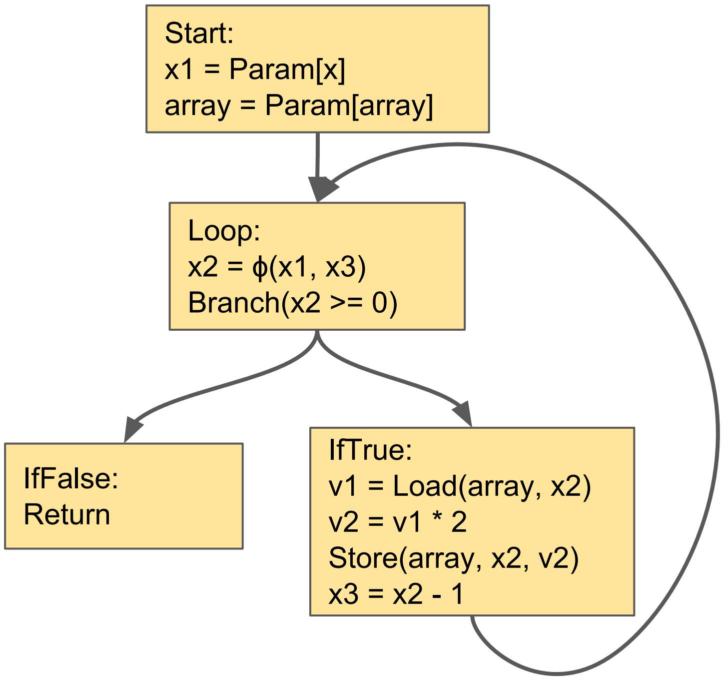
Turbofan - A Sea of Nodes Compiler

```
function double(x, array) {
 while (x >= 0) {
   array[x] *= 2;
   x--;
 }
}
```

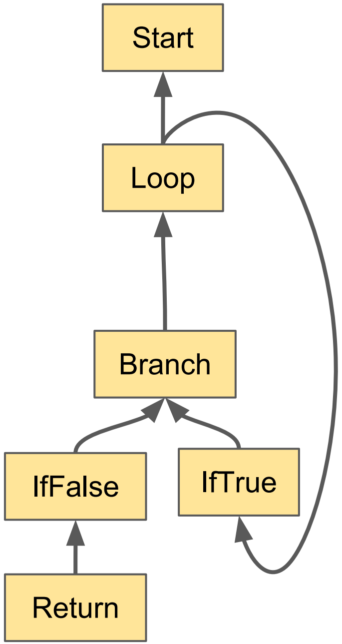


Turbofan - A Sea of Nodes Compiler

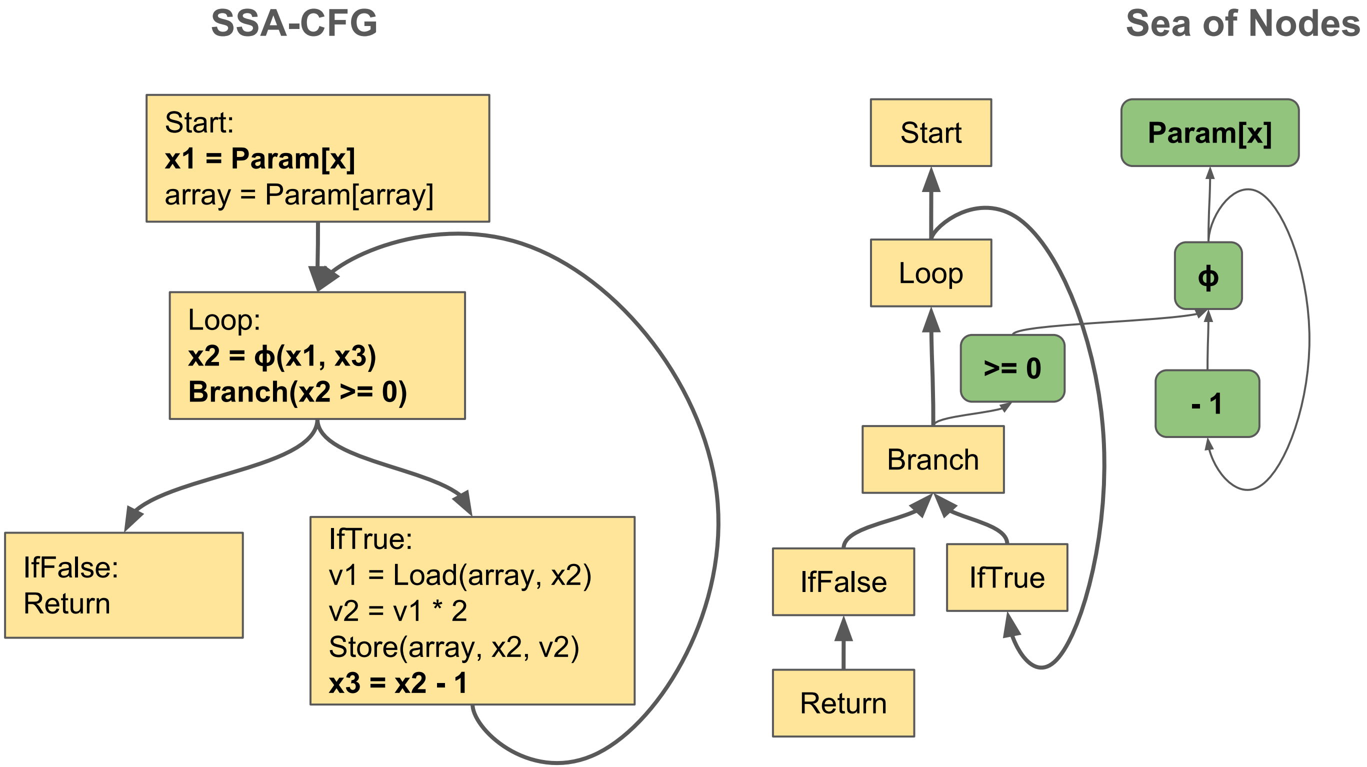
SSA-CFG



Sea of Nodes

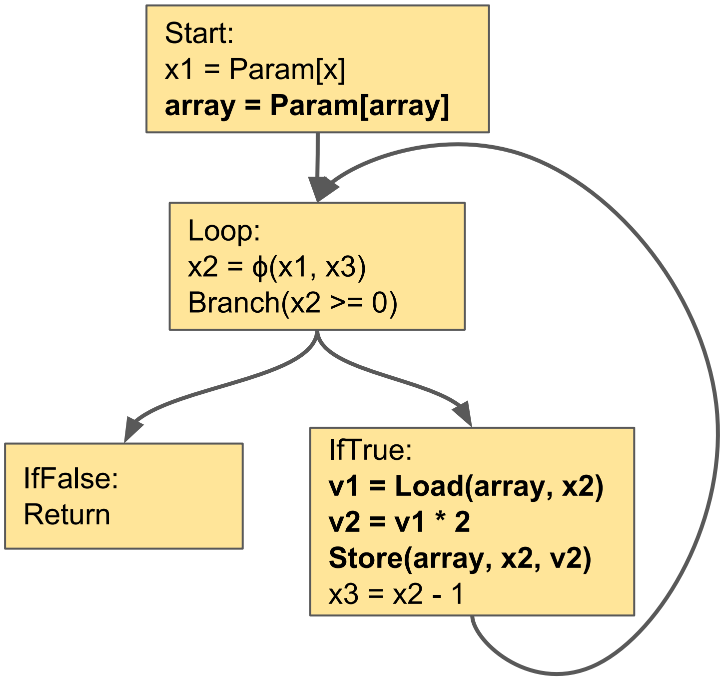


Turbofan - A Sea of Nodes Compiler

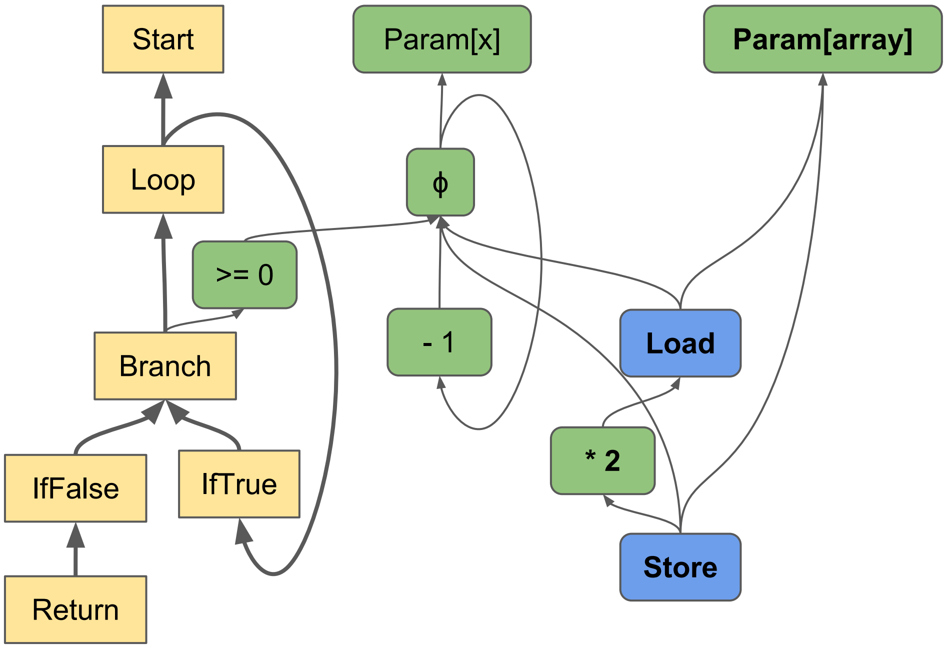


Turbofan - A Sea of Nodes Compiler

SSA-CFG

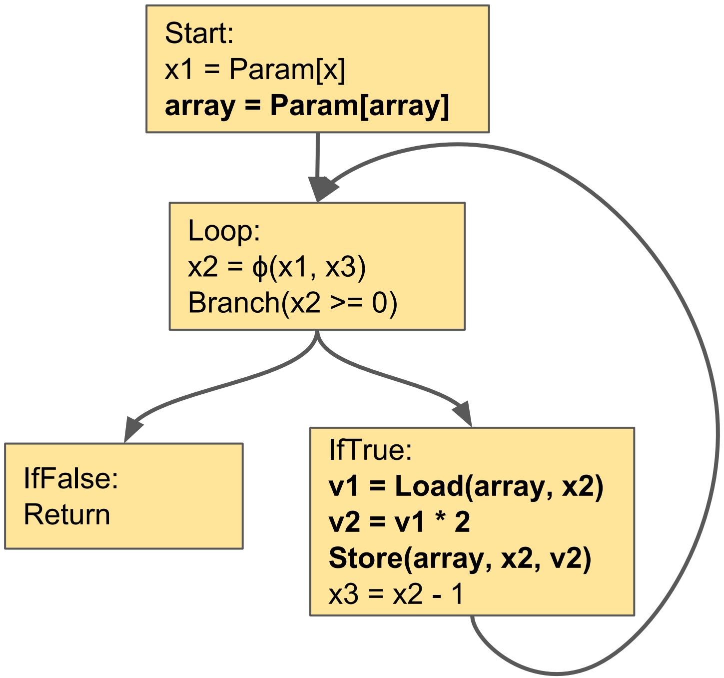


Sea of Nodes

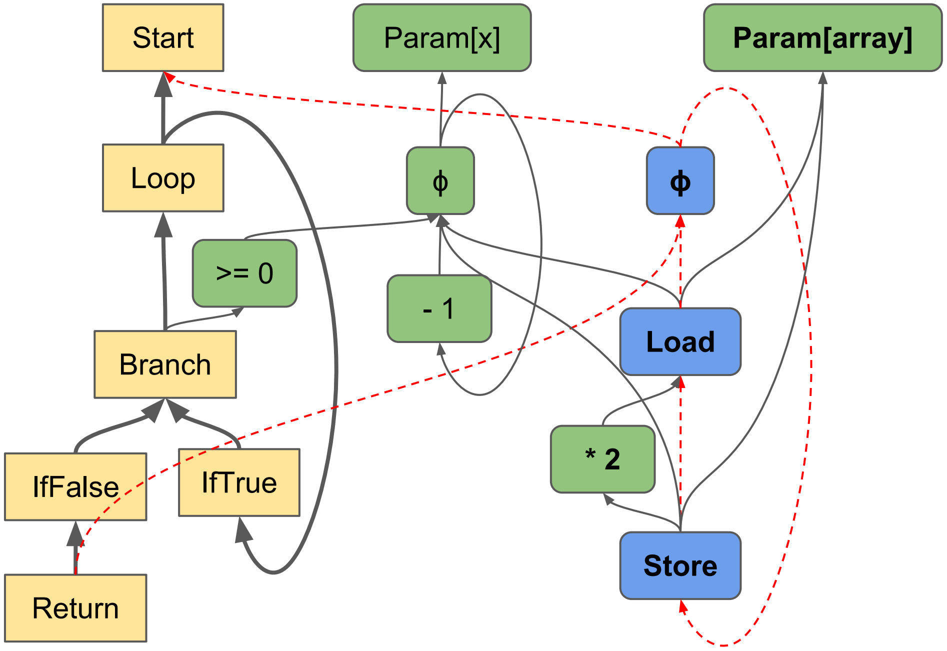


Turbofan - A Sea of Nodes Compiler

SSA-CFG



Sea of Nodes



Why Sea of Nodes?

Advantages

Disadvantages

Why Sea of Nodes?

Advantages

- All dependencies are explicit.

Disadvantages

- All dependencies have to be explicit.

Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.

Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.

Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.

Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.
- Hard to replace pure nodes with local effect or control.

Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.
- Hard to replace pure nodes with local effect or control.
- Reductions operate on nodes instead of blocks.

Escape Analysis:
Removing Temporary Objects

```
function diagonal(a) {
  return abs({x:a, y:a});
}
```

⇒

```
return Math.sqrt(a*a + a*a);
```

```
function abs(v) {
  return Math.sqrt(v.x*v.x + v.y*v.y);
}
```

How do we remove this object allocation?

Step 1: Inlining

```
function diagonal(a) {
    let v = {x:a, y:a};
    return abs(v);
}

function abs(v) {
    return Math.sqrt(v.x*v.x + v.y*v.y);
}
```

```
function diagonal(a) {
    let v = {x:a, y:a};
    return Math.sqrt(v.x*v.x + v.y*v.y);
}
```

Step 2: Replace Field Accesses

```
function diagonal(a) {
   let v = {x:a, y:a};
   return Math.sqrt(v.x*v.x + v.y*v.y);
}
```

→

```
function diagonal(a) {
   let v = {x:a, y:a};
   return Math.sqrt(a*a + a*a);
}
```

Step 3: Remove the unused allocation

```
function diagonal(a) {
    let v = {x:a, y:a};
    return Math.sqrt(a*a + a*a);
}
```

```
function diagonal(a) {

    return Math.sqrt(a*a + a*a);
}
```

It's not always that easy...

Writing to Fields

```
function write_field(x) {
 let o = {a: 5};
 while (x > 0) o.a += x--;
 return o.a;
}
```

⇨

```
function write_field(x) {
 let o_a = 5;
 while (x > 0) o_a += x--;
 return o_a;
}
```

Replace object fields with a local variable.

Nested Objects

```
function nested_objects(b) {
 let o = {a: {x: 5}};
 o.a = {x: 7}
 return o.a.x;
}
```

→

```
function nested_objects(b) {


 return 7;
}
```

Limitations

Escaping Objects

```
function escaping_object(foo) {
 let o = {};
 foo(o);
}
```

When we can’t inline foo, then we cannot dematerialize o because foo could do anything with it.

Index Access

```
let l = [1,2,3];
let sum = 0;
for(let i = 0; i < 3; ++i) sum += l[i];
```

Index access requires linear memory.

Thus we have to materialize `l`.

Dynamic Object Identity

```
function object_identity(b) {
 let o1 = {x: 5};
 let o2 = {x: 7};
 (b?o1:o2).x = 1;
 return o1.x;
}
```

This computation uses the object identity of o1 and o2.

It's impossible to map this to local variables: Local variables don't have identity.

In principle, this could be solved with stack-allocation. (Java VMs do this.)

The Magic of Deoptimization

```
function harmless(copy) {}

function foo(x) {
  let copy = {};
  copy.a = x + 1;
  harmless(copy);
}

function evil(copy) {
  global = copy;
}
foo({valueOf: () => harmless=evil});
```

?

```
function foo(x) {
  x + 1;
}
```

While executing foo, the temporary object might suddenly escape.

Monkey patching can destroy any optimization, while the optimized code is running.

The Magic of Deoptimization

```
function harmless(copy) {}

function foo(x) {
  let copy = {};
  copy.a = x + 1;
  harmless(copy);
}
```

```
function foo(x) {
 if (typeof x != 'number')
   %Deoptimize();
}
```

create object copy

When deoptimizing, we have to re-create dematerialized objects.

Escape Analysis needs to store the state of dematierialized objects at each deoptimization point.

The Algorithm

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

global

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	null

for every effectful node

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	null
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	null
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
  a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

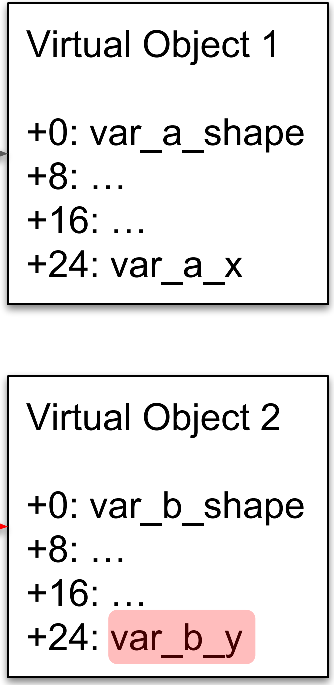
+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
  a.x.y = i;
}
foo();
return a.x.y;
```



Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	5

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
  a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	i

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	Φ(5,i)

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x.y;
```

Remember Deoptimization Data

+0: shape of {x:?}
+8: …
+16: …
+24:

+0: shape of {y:?}
+8: …
+16: …
+24: Φ(5,i)

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	Φ(5,i)

Escape Analysis in Turbofan

~~let a = {x: null};~~
~~let b = {y: 5};~~
~~a.x = b;~~
if (i > 10) {
 ~~a.x.y = i;~~
}
foo();
return ~~a.x.y~~; Φ(5,i)

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Current Variable Value

var_a_shape	shape of {x:?}
var_a_x	b
var_b_shape	shape of {y:?}
var_b_y	Φ(5,i)

Escape Analysis in Turbofan

```
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
 a.x.y = i;
}
foo();
return a.x;
```

The inner object escapes!
Repeat analysis from its allocation point.

Virtual Object 1

+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Virtual Object 2

+0: var_b_shape
+8: …
+16: …
+24: var_b_y

Challenges

Repeat When Escaping

- At any point, we might notice an object escapes.
- This invalidates all analysis steps using this object.
- But how to restore the previous state?

Solution:

- Separate analysis from graph mutation.
- Only do graph mutation once the analysis reached a fixed point.
- Track node replacements while analyzing.

How to Store the Variable State

- In a CFG: One map per basic block, updated imperatively when traversing the block
- In an unscheduled graph: One map per effectful node.

This is expensive! Solution: A purely functional map:

  - Copy: O(1)
  - Update/Access: O(log n)

This can be achieved with any tree-based map datastructure.
We chose a hash-tree.

Summary

- Escape analysis avoids allocating temporary object.
- V8 can dematerialize objects despite deoptimization.
- Limits of escape analysis are: escaping uses, index access, and using the object identity dynamically.
- Implementing escape analysis on an unscheduled graph is more challenging: All object fields need to be tracked for all effectful nodes.
- Purely functional maps can solve this issue without increased complexity.