High performance JavaScript with V8

Florian Schneider
Software engineer
Google Aarhus, Denmark
March 6, 2012

Thanks to
Kevin Millikin and V8 team

Outline

V8 basics:

- Inline caching
- Hidden classes
- Object model
- Adaptive optimization

Inline caching

- Prior work on Self, Scheme, Smalltalk already in the 80s
  - We stand on the shoulders of giants
- Main optimization in Chrome 1
  - Still an integral part in current V8
- Method invocation, property access faster
  - Also used for arithmetic, compare operations

Google

JavaScript objects

- Prototype-based inheritance
- Dictionary storing key-value pairs
- Key is always a string.

```
> var o = {x:3}
> o.x
3
> o["x"]
3
>
```

Odd cases with JavaScript

- Array out-of-bounds loads
  - Require lookup in the prototype
  - If no property found, return undefined

```
var a = [„hest“, „fisk“]
Object.prototype[7] = „oops“;
print(a[7]);
> oops
```

Example: property load

```
function MyObject(x) {
  this.X = x;
}
function getX(obj) {
  return obj.X;
}
var o = new MyObject(3);
print(getX(o));
```

Property load procedure

- Simple approach:
  1. Query object if property exists
     - If found, load value from the property
  2. If not, follow up the prototype chain
     - Query protypes until property found
     - If found, load value from the property
  3. If not found, return undefined

→ Very expensive!

Basic idea: caching

- Approach 1: Hash table
  - <object type, property name> as key to quickly find location of a property
  - Faster than the repeated lookup procedure, but...
- Approach 2: Inline cache
  - Cache result of last lookup in the instruction stream
  - Implement cache using self-modifying code
  - No extra data structure needed

Google

Requirements

1. Need an efficient representation
2. Need an efficient way of testing if and where a property exists.

- Objects have dictionary semantics
- Hash table implementation expensive
  - Lookup procedure expensive (prototype!)

Observations

- getX only ever sees MyObject instances
- MyObject instances have an X property
- X is a normal field
- Knowing that obj is a MyObject can make loading obj.X faster!

How can we test this and load the property really efficient?

Hidden classes

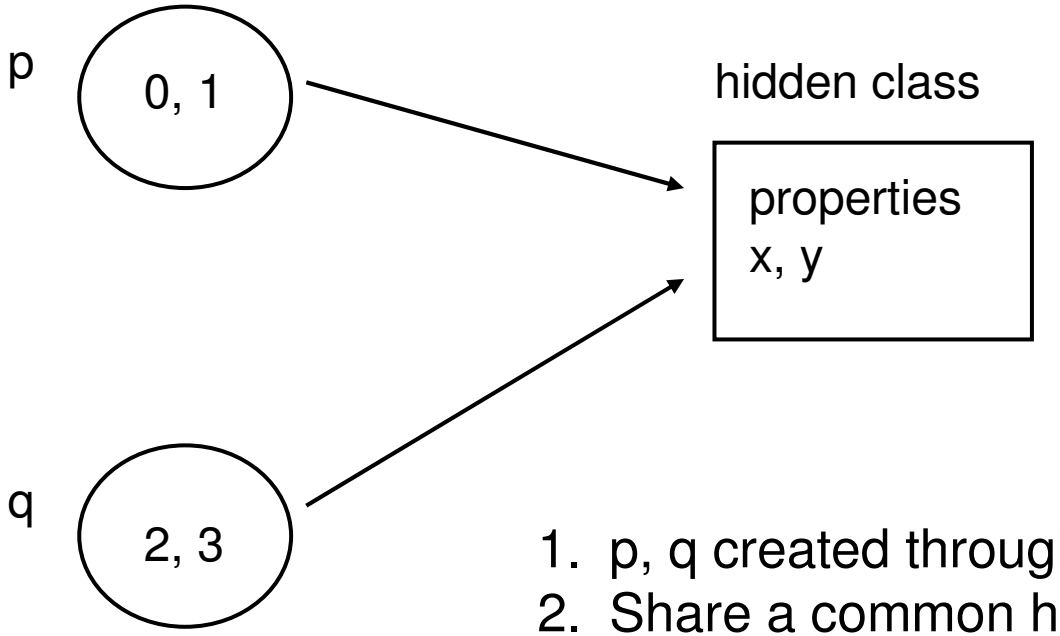
- In V8 terms aka. „*maps*“
- Basically a object layout description
  - Created automatically by V8
  - Consists of property descriptors
  - Key/value pairs (name/attributes)
- Form the prototype hierarchy
  - Implement prototype chain
  - Maps points to prototype object

Example

- JavaScript prototype-based
  - No static class structure
  - Properties added by assignments

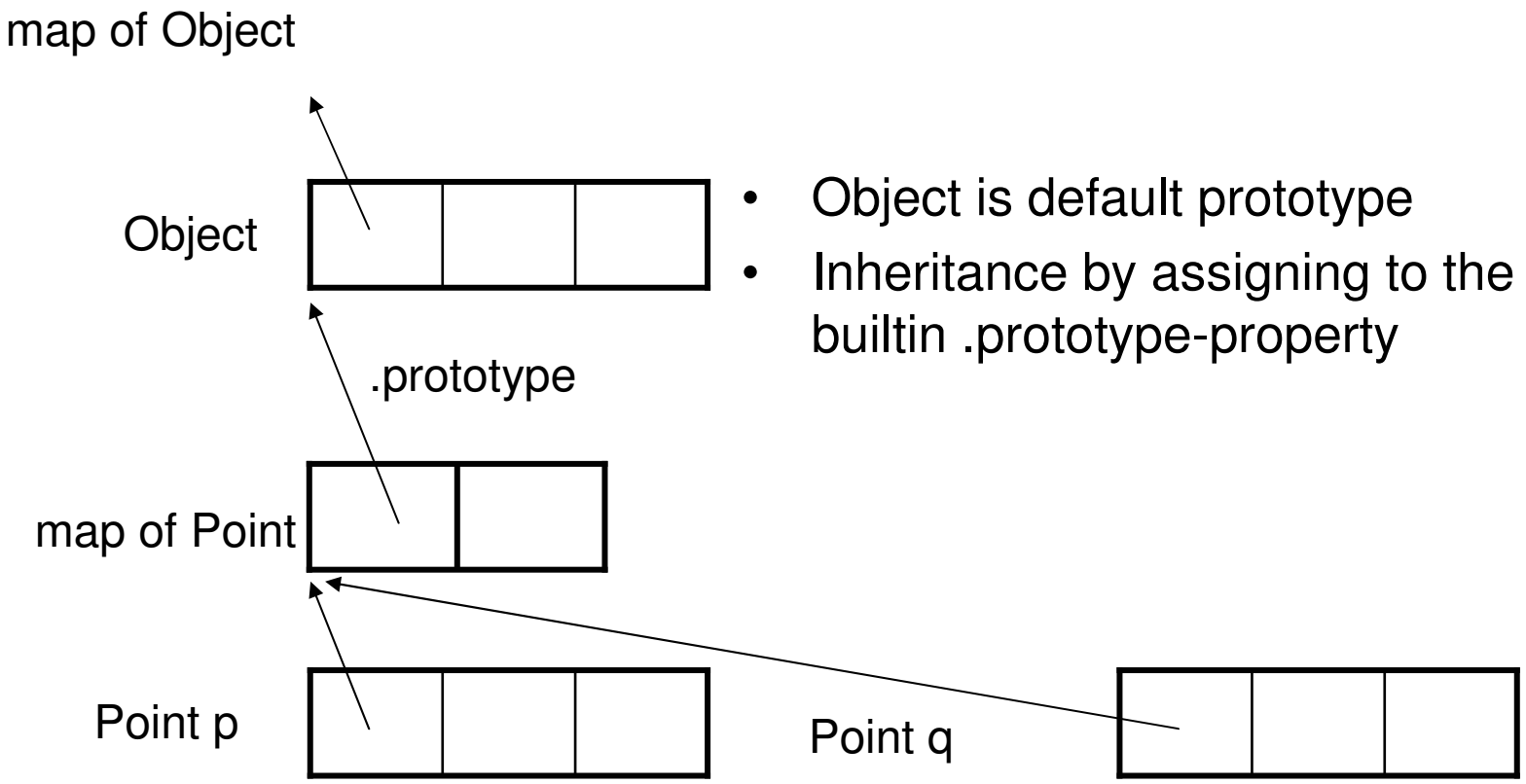
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p = new Point(0, 1);
var q = new Point(2, 3);
```

Example



1. p, q created through the same constructor
2. Share a common hidden class

Prototype chain

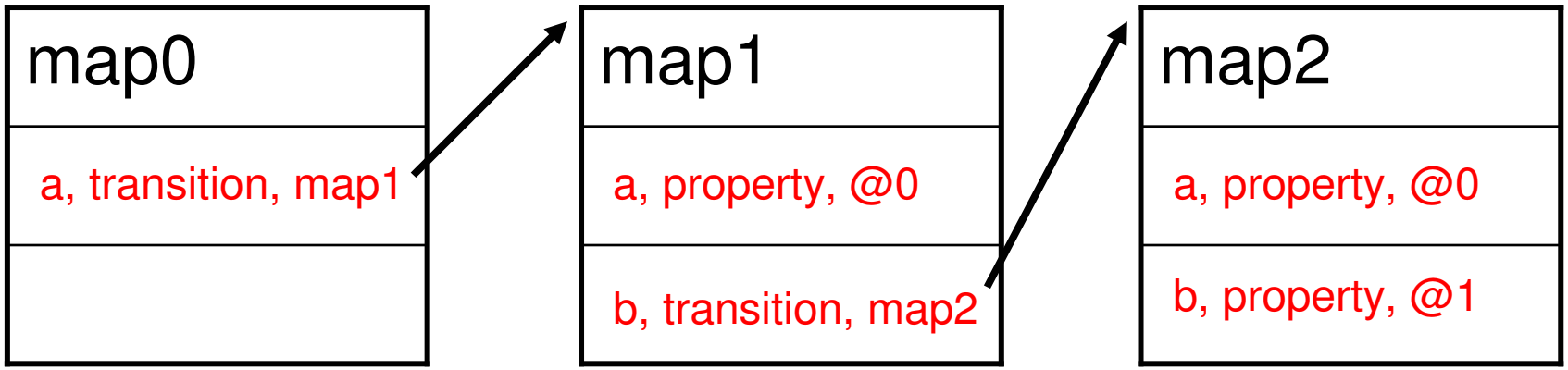


- Object is default prototype
- Inheritance by assigning to the builtin .prototype-property

Map transitions

- Triggered by adding properties
  - Whenever object changes shape
  - Needs a new map describing the layout
- Compute new map
  - Update property descriptors
  - Maps connected via transition-tree
- Allow easy sharing of maps
  - Objects with the same shape share maps

Map transition example



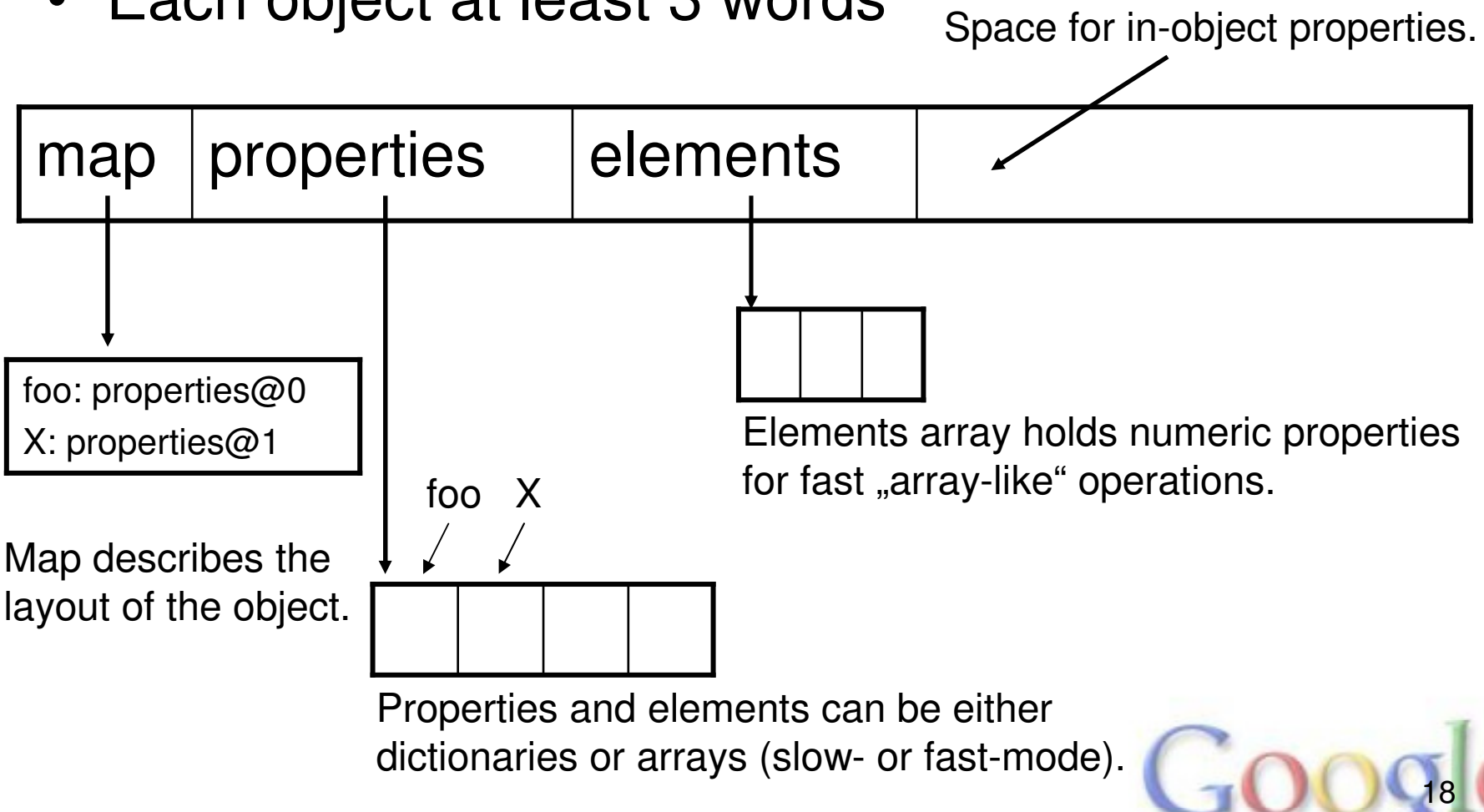
- `var o = {};`
- `o.a = 0;`
- `o.b = 0;`

Q: What are maps good for?

- A: Make property access and method invocations really efficient

V8 object model in detail

- Each object at least 3 words



Inline caching (IC)

- One element cache of the last map seen
- Cached map is stored inline in the code
- If we encounter the cached map (hit), we already know where to find the property
- If not (miss), some additional work required (i.e. update cache)

Google

Inline cache in assembly

```
      mov  ecx, „x“
      mov  eax, obj
      cmp  [eax + kMapOffset], <cached map of obj>
      jne  miss
      mov  eax, [eax + kPropertiesOffset]
      mov  eax, [eax + <cached offset of property x>]
      jmp  done
miss:
      call IC_Miss
done:
```

Uninitialized state → „Monomorphic“ state

Monomorphic state

```
      mov  ecx, „x“
      mov  eax, obj
      cmp  [eax + kMapOffset], <cached map of obj>
      jne  miss
      mov  eax, [eax + kPropertiesOffset]
      mov  eax, [eax + <cached offset of property x>]
      jmp  done
miss:
      call IC_Miss
done:
```

Cache hit case!

But what happens if we encounter a new map?

Polymorphic sites

- Change to a new monomorphic state?
  - Problem with true polymorphic sites
  - e.g sequence with maps ABABABA...
  - Switching back and forth is expensive
- Add new map to the cache
  - Grow cache to two elements?
- Can‘t easily insert a new map-compare
  - No space for patching in new code

Alternative IC implemenation

Compile a small routine checking for 1..n maps and returning the result value

- No inline code, patch call site instead
- V8 uses (mostly) this approach
  - Trade-off: space/time
  - Code size smaller: good for page load time
  - Call overhead

Google

Polymorphic IC in assembly

```
mov   ecx, „x“
mov   eax, obj
call  IC_Polymorphic_x
```

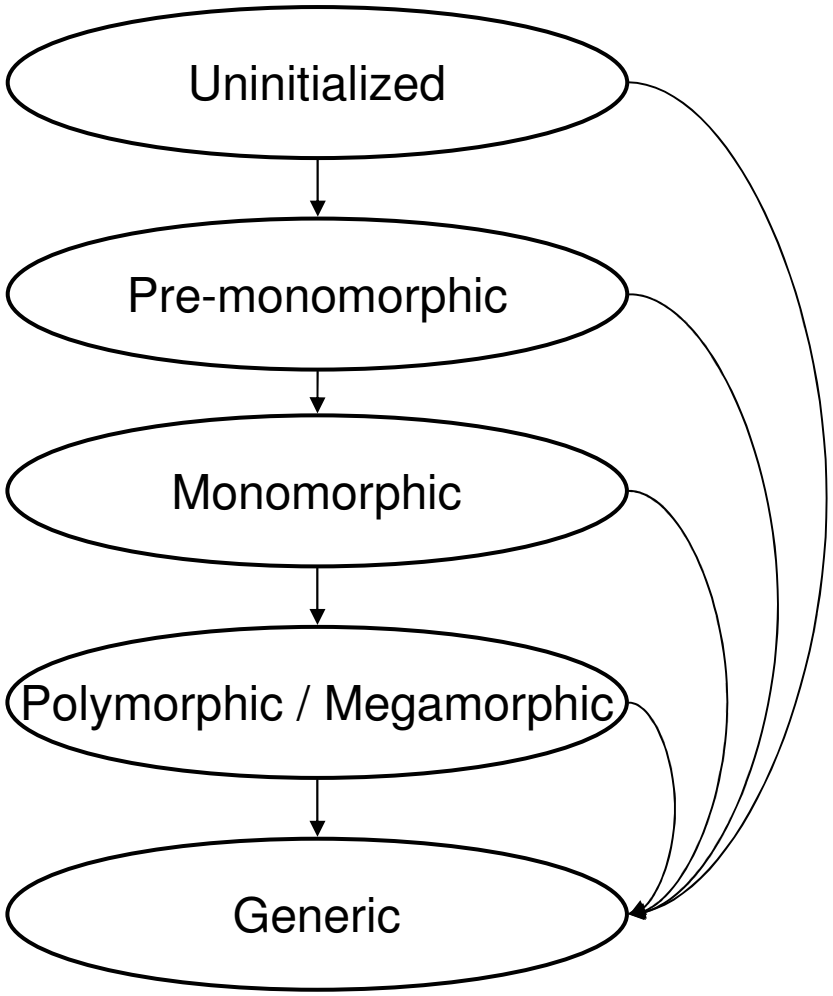
```
IC_Monomorphic_x:
    cmp   [eax], <map>
    jne   miss
    mov   eax, [eax + 4]
    mov   eax, [eax + <x_offset>]
    ret
miss:
    jmp IC_Miss
```

```
IC_Polymorphic_x:
    cmp   [eax], <map1>
    je    check2
    mov   eax, [eax + 4]
    mov   eax, [eax + <x_offset1>]
    ret
check2:
    cmp   [eax], <map2>
    jne   miss
    mov   eax, [eax + 4]
    mov   eax, [eax + <x_offset2>]
    ret
miss:
    jmp IC_Miss
```

Megamorphic sites

- Some places highly polymorphic (n >> 4)
- V8 uses a hash table instead of a long sequence of compares
- Cache lookup results for <name, map> pairs

V8 IC states

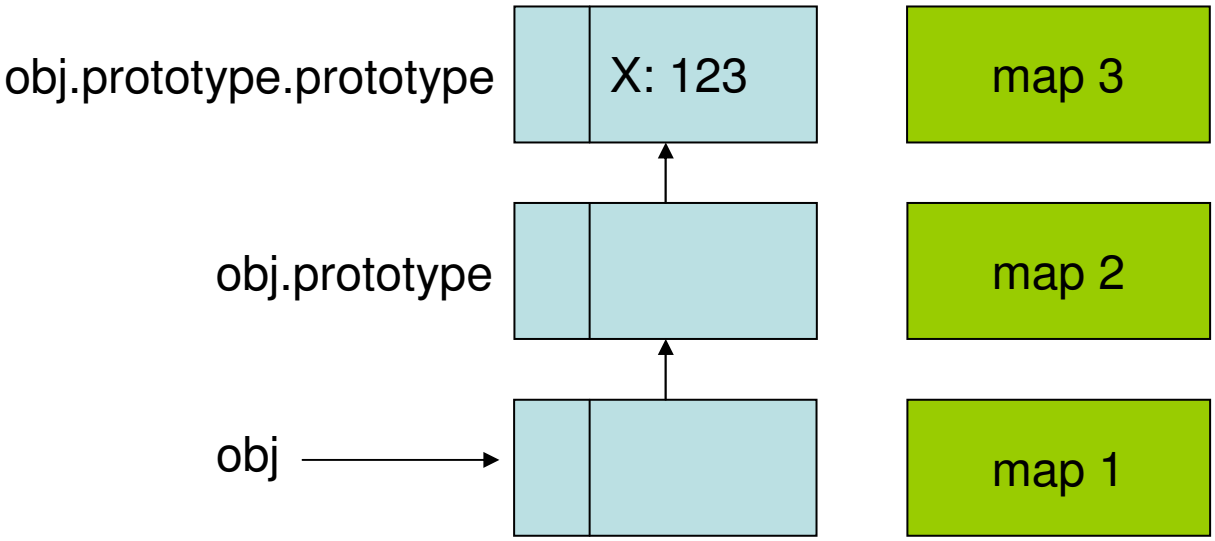


- Uninitialized: Start
- Premonomorphic
  - Avoid patching code that is run only once

- Generic slow case not handled by IC: e.g.
  - Non-existing element
  - getter/setter

Access inherited properties

- If holder of a property is a prototype, IC must check maps of prototype chain.



IC map check procedure

1. Check map of obj
2. Check map of obj.prototype
3. Check map of obj.prototype.prototype
4. Load property from cached location

- # map checks ~ prototype chain depth

ICs summary

- Fastest for monomorphic sites
- Performance varies with the degree of polymorphism
- Inherited properties/methods much more expensive (prototype chain)

Questions

Google™

V8 Crankshaft

- New adaptive compilation system for V8
- Designed and implemented in Aarhus
- Announced in December 2010
- Shipped as part of Chrome 10
  - Improved and updated frequently since then
- Enabled us to shake up JavaScript performance (again)

Crankshaft approach

1. Spend time optimizing hot functions, not cold functions

   – Result: high peak performance AND fast startup time (shorter page load time)
   – Don’t waste time optimizing code run once

Crankshaft approach

2. Optimize based on runtime type feedback
   - Hints for the expected types of objects
   - Allows better elimination of dynamic checks like smi-checks or map-checks

Crankshaft approach

3. Enable new optimizations

- Many classic compiler optimizations for JavaScript
  - Common subexpression elimination
  - Loop-invariant code motion
  - Function inlining
  - Global register allocation

2. Type feedback

- ICs contain runtime type information
  - Maps seen
  - Types in arithmetic operations
- Mine IC call sites from non-optimized code
  - Use as type hints for optimized code

Google

Type feedback example

```
function dot(p, q) {
  return p.x * q.x + p.y * q.y;
}
```

In this case:

- ICs for p.x, q.x, p.y q.y monomorphic
- Cached map is map of Point
- IC for * and + record operand types
  - Possible states: smi, double, string, other non-number

High-level IR for dot function

```
          check-map p, Point
t1 =      load p.x
          check-map q, Point
t2 =      load q.x
          check-number t1
          check-number t2
t3 =      t1 * t2
          check-map p, Point
t4 =      load p.y
          check-map q, Point
t5 =      load q.y
          check-number t4
          check-number t5
t6 =      t4 * t5
return    t3 + t6
```

- Type checks as predicted by the type feedback

High-level IR for dot function

```
          check-map p, Point ──→ ?
t1 =      load p.x
          check-map q, Point ──→ ?
t2 =      load q.x
          check-number t1 ──→ ?
          check-number t2 ──→ ?
t3 =      t1 * t2
t4 =      load p.y
t5 =      load q.y
          check-number t4 ──→ ?
          check-number t5 ──→ ?
t6 =      t4 * t5
return    t3 + t6
```

- Type checks as predicted by the type feedback
- Checks can be eliminated
  - No side effects!

But: Something missing...

- What if a check fails?

Deoptimization

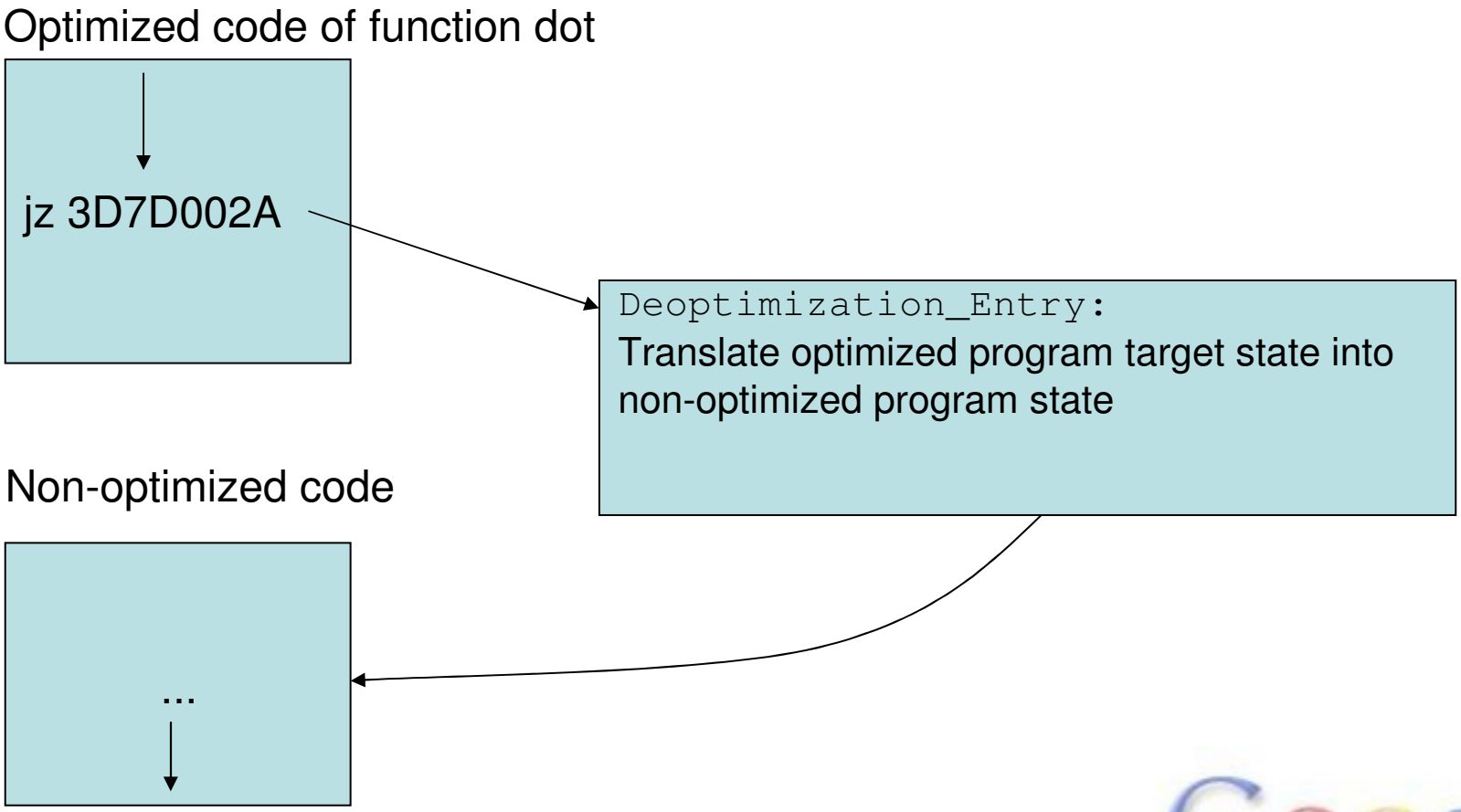
- If a check in the optimized code fails, continue execution in non-optimized code (deoptimize)

Example code

Assembly code for loading p.x, q.x

```
0    mov eax,[ebp+0xc]              ;; load p
1    test al,0x1                    ;; smi-check p
2    jz 3D7D002A                    ;; deoptimization bailout 1
3    cmp [eax-1],0x19f2461          ;; map-check p: 019F2461 <Map>
4    jnz 3D7D0034                   ;; deoptimization bailout 2
5    mov ecx,[eax+11]               ;; load p.x (in-object
6    mov edx,[ebp+0x8]              ;; load q
7    test_b edx,0x1                 ;; smi-check q
8    jz 3D7D003E                    ;; deoptimization bailout 3
9    cmp [edx+0xff],0x19f2461       ;; map-check q: 019F2461 <Map>
10   jnz 3D7D0048                   ;; deoptimization bailout 4
11   mov ebx,[edx+0xb]              ;; load q.x
```

Control-flow on deoptimization



Deoptimize HowTo

- Find the point to jump to
  - Check instructions need info
- Enter a state so the non-optimized code can continue
  - Restore registers, activation stack
  - Unfold optimized stack frame into 1..n non-optimized frames (inlining)
  - Box untagged values (numbers)

Deoptimiztion translation

- Non-optimized code is a stack machine
  - All locals, parameters on the stack
  - No register allocation
- Simulate expression stack state while compiling the optimized code
  - Push/pop
  - Assign vales of locals, parameters
- At each check-instruction we save a copy of the current state

After deoptimizing

- Non-optimized code runs
- Optimized code discarded
- Once function gets hot again
  - May be re-optimized with new, updated typefeedback
  - Optimizing compiler „learns“ about new types

V8 Crankshaft summary

- V8 Crankshaft brings performance of JS much closer to the level of statically typed languages (like Java)
  - Many known optimizations done for the first time for JavaScript
- Page load time + peak performance
- New apps possible
  - Playing Angry Birds in the browser, etc.

Google

Q&A

Google