Compiling Dart to Efficient Machine Code

Dr. Florian Schneider
Software engineer
Google Aarhus, Denmark
April 19, 2012

Introduction

- Working at Google Denmark on
  - Dart: New programming language
  - V8: Chrome's JavaScript engine
- Graduated from ETH in 2009
- Lots of challenging compiler-related work

Dart Introduction

- Structured programming for web apps
- Dynamically typed w/ optional static types
- Class-based
- Libraries
- Isolates
- Futures
- SDK Tools
  - Editor, analyzer, debugger

Overview

- Dart intro
- Dynamic typing
- Inline caching
- Object model
- Optimizations

Dot example

```
class Dot {                                   // Class declaration
  num x, y;                                   // Field declaration
  Dot(this.x, this.y);                        // Construction (short-hand form)
  String toString() => "($x,$y)";             // Method
}                                             // Short-hand forms: => and $

void main() {
  var p = new Dot(3, 4);                      // Constructor call
  var q = new Dot(5, 6);
  print(p);                                   // Built-in global function print
}
```

Dart classes

- Static single heritance: `extends`
  - Interface inheritance: `implements`
- Fixed number of members
  - Fields, methods
- All values instance of `Object`
  - Dot class overrides `toString` method
- Accessing non-existing member
  - NoSuchMethodError

Optional types

```
class A {
  m() { print("A.m"); }
}

class B {
  m() { print("B.m"); }
}
```

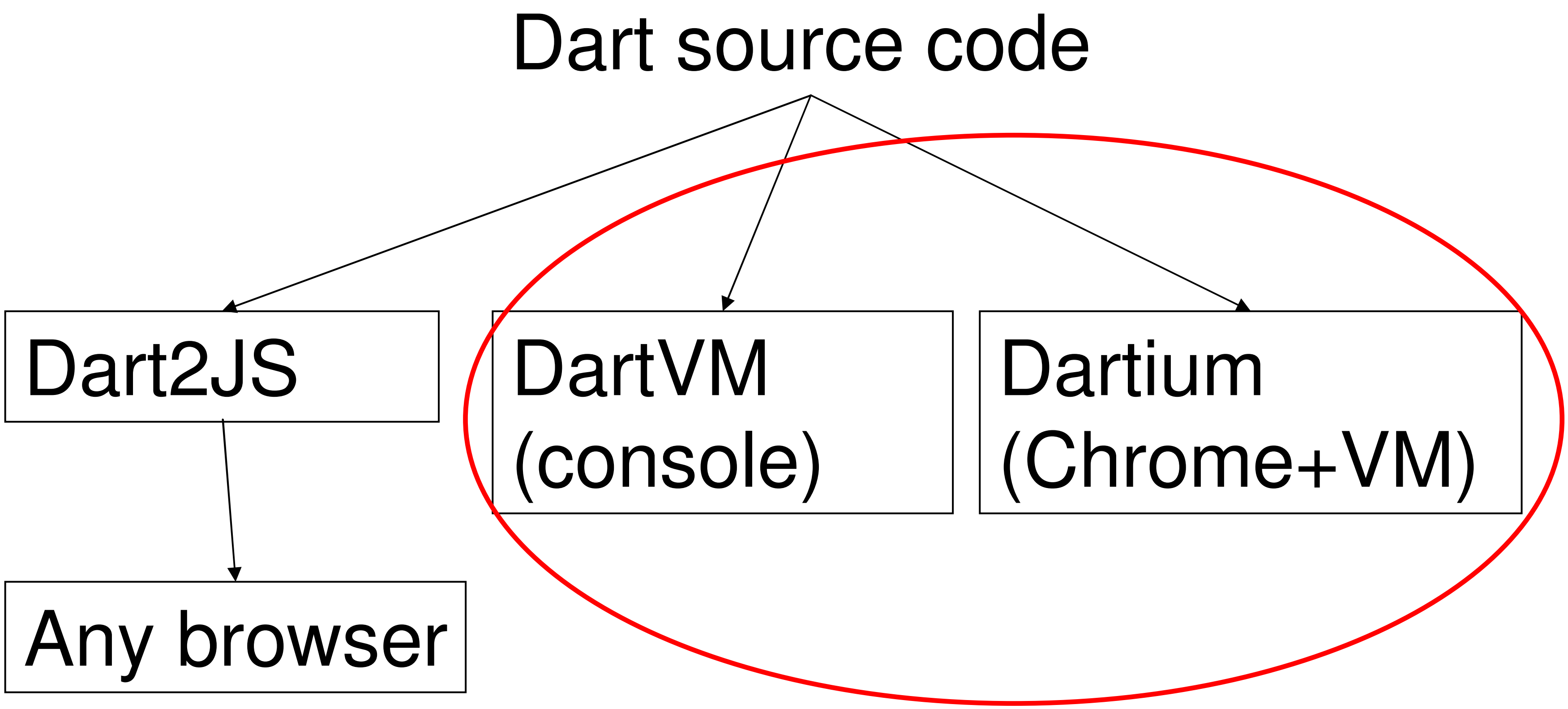
```
         A x
void f(var x) {
  x.m();
}

main() {
  f(new A());
  f(new B());
}
```

- Keyword `var` (or no type) allows everything
- Types optional
- Asserted at run-time (can run without assertions)

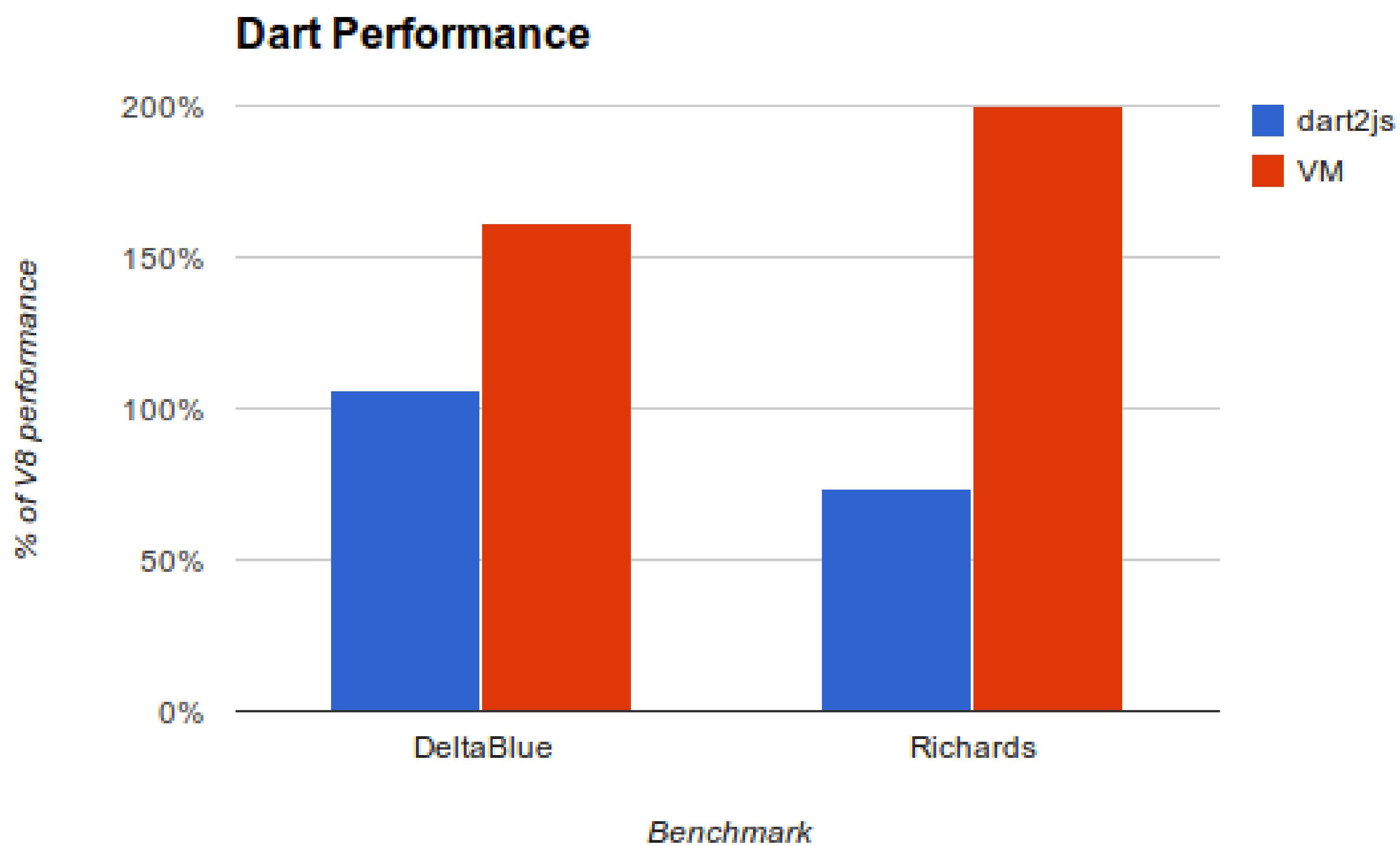
Optional types

- Useful to specify programmer intent
- Useful for development tools
  - Issue static warnings
  - Smart auto-completion
- Catch errors earlier
  - Get TypeError, instead of a later NoSuchMethodError

Dart execution



Performance comparison



Method invocation

```
void f(var x) {
  x.m();
}
```

- Dynamic typing requires run-time lookup
  - Run-time error is method m not found
- Everything is an object
  - Expression `f(123)` valid, but will throw
  - But `123.toString()` => `"123"`

Value representation

- Align all addresses to 4 bytes (for 32-bit arch)
  – Required for RISC (ARM), good idea on ia32 as well
- Values are 1 word, containing either:
  – Small integer (smi), tag == 0

31 bit signed int	0

  – Object reference, tag == 1

30 bit pointer	0	1

Tagged pointers

- Efficient addressing:
  - Object header `*(tagged_ptr - 1)`
- Efficient integer operations on smis
  - No indirection / boxing
- Quick conversion to real integers from smi
  - Using >> and <<
- Only limited range (-2^30...2^30-1)
  - Larger: heap allocated number object

Function f in pseudo-code

```
t0 := LoadClass(x)
t1 := t0.LookupMethod("m")
if (t1 != NULL)
    call t1.code()
else
    throw NoSuchMethodErr(t0,"m")
```

Cheap

Expensive!

Cheap

Speeding up lookup

- Observation: If type of parameter x stable
  - Caching lookup results very beneficial
- Common technique: Inline caches
  - Long history: Smalltalk, Self, Scheme, JavaScript, Dart

Inline cache in V8

```
mov  ecx, „m“
mov  eax, obj
call IC_Polymorphic_AB_m
```

```
IC_Polymorphic_A_m:
   test eax, 1
   jz   miss
   cmp  [eax - 1], class_id_A
   jne  miss
   jmp A_m
miss:
   jmp IC_Miss
```

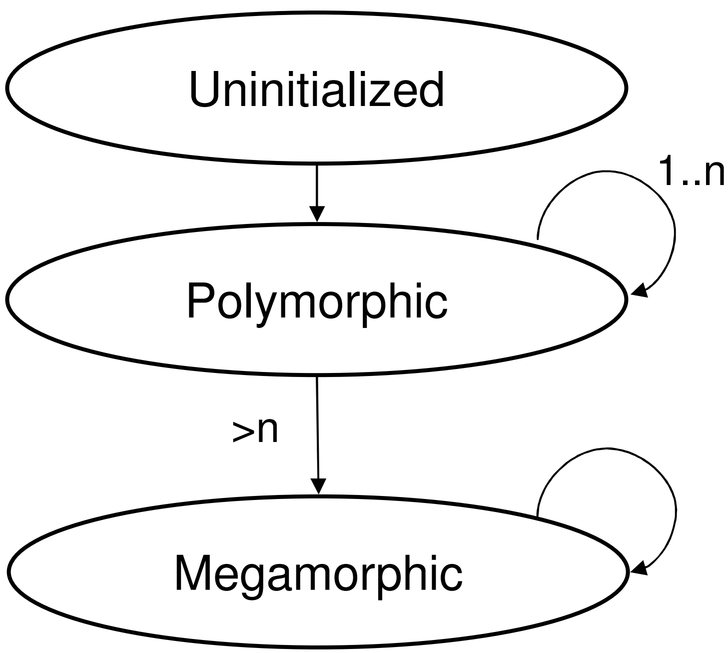
```
IC_Polymorphic_AB_m:
   test eax, 1
   jz   miss
   cmp  [eax - 1], class_id_A
   je   check2
   jmp  A_m
check2:
   cmp  [eax - 1], class_id_B
   jne  miss
   jmp B_m
miss:
   jmp IC_Miss
```

IC cache miss handler

IC_Miss(obj, name, state)

1. Lookup(obj, name)
2. UpdateCache(obj, name, state)
   - If state changes: Patch call site with new IC
3. Call target and return

IC states



- Megamorphic: Hash-based lookup
  - after limit on # of classes

Inline cache in Dart VM

- Array-based IC
- Associate array as IC with each call-site
  - Array of <arg-types, target, count>
  - Dispatch on receiver (first argument)
  - Other arguments types collected as needed
- IC array attached to each call-site
- No need for code patching

Array-based IC

- Slower than embedding cache in code
- Focus on collecting detailed information at each call site
  - Argument types (not just receiver class)
  - Exact invocation counts

Inline caching summary

- Basic wide-known optimization technique
  - Long history
  - Employed in both Dart VM and V8
- Need class-based system
  - Not directly applicable to JavaScript
  - V8 uses "hidden classes"

Excursion: JavaScript

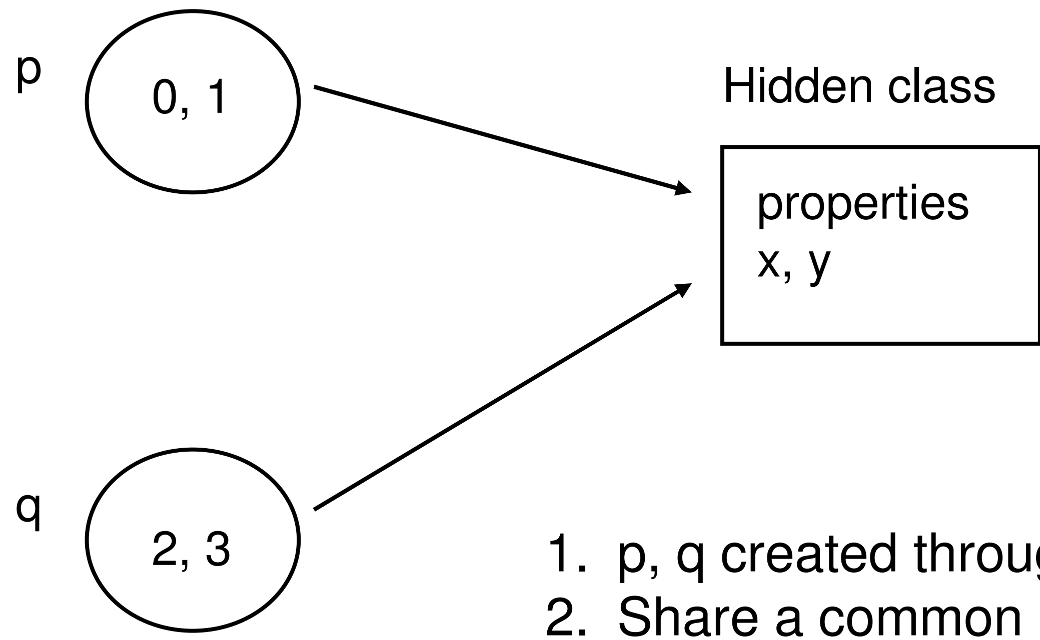
- Basic concept of IC similar, but
- JS has no classes
- Instead the VM creates "hidden classes" (aka. maps)
- Object layout a lot more complicated
  - Various tricks to speedup property access

Example

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

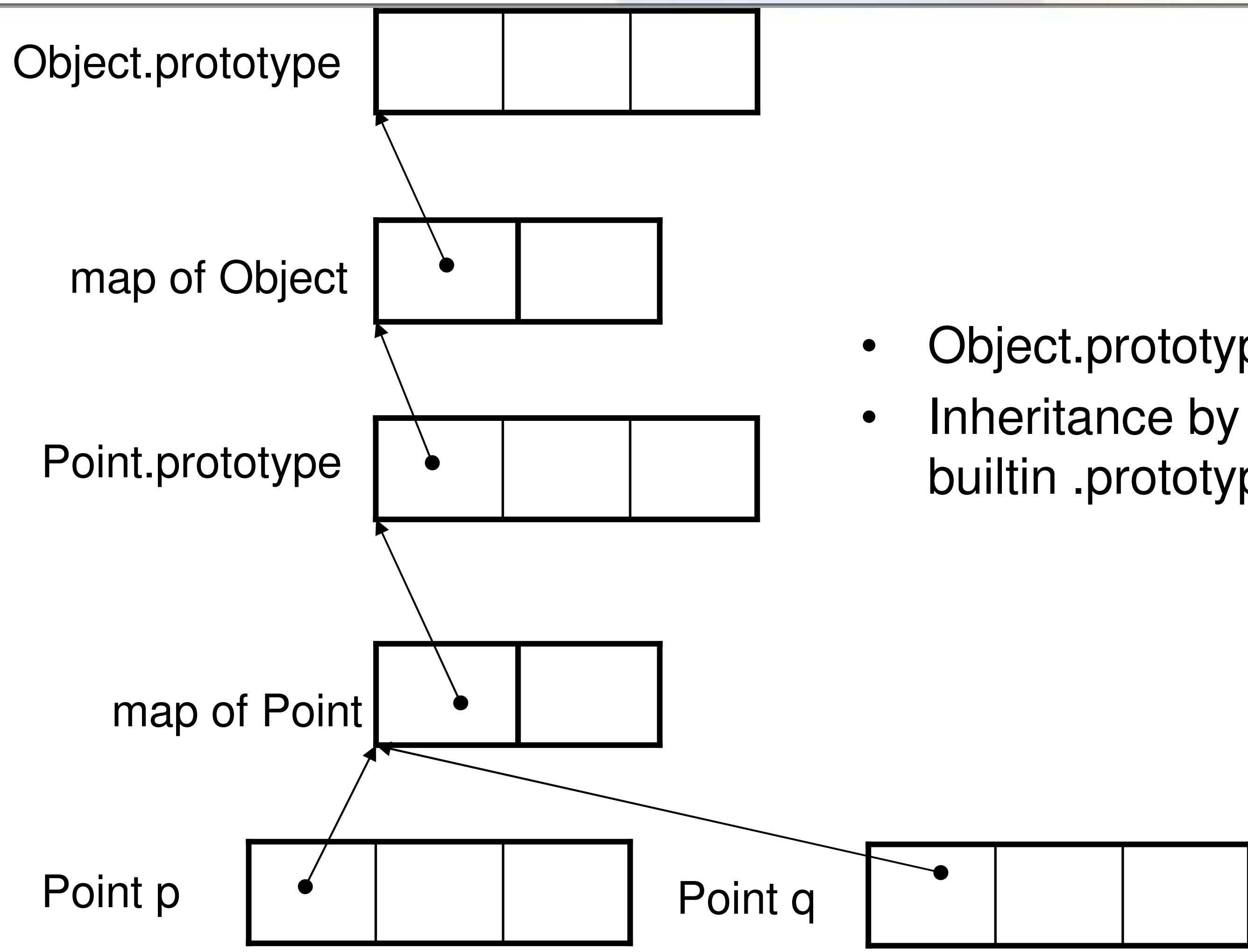
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p = new Point(0, 1);
var q = new Point(2, 3);
```

Example



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

Prototype chain

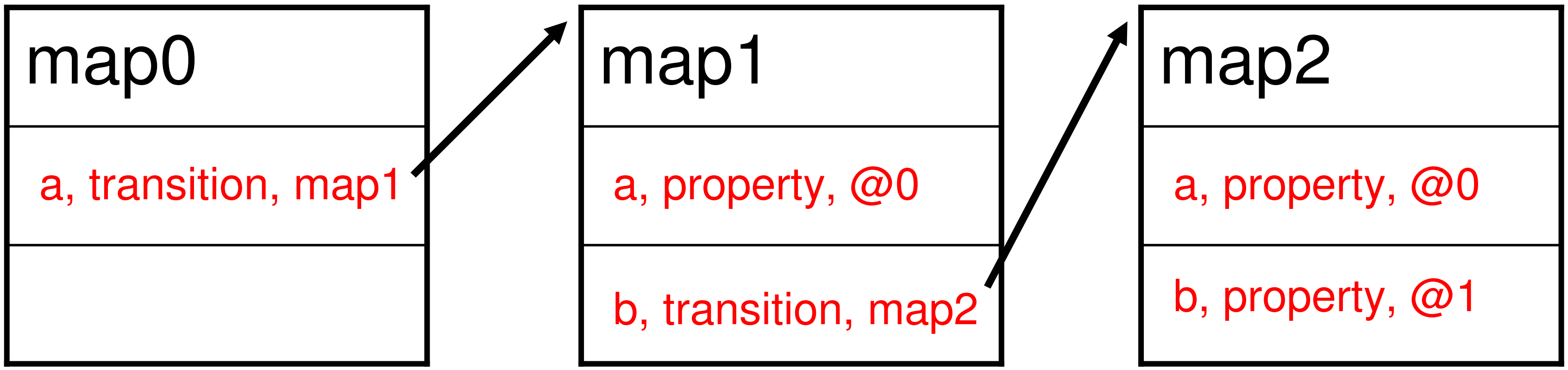


- Object.prototype at the top
- Inheritance by assigning to the builtin .prototype-property

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



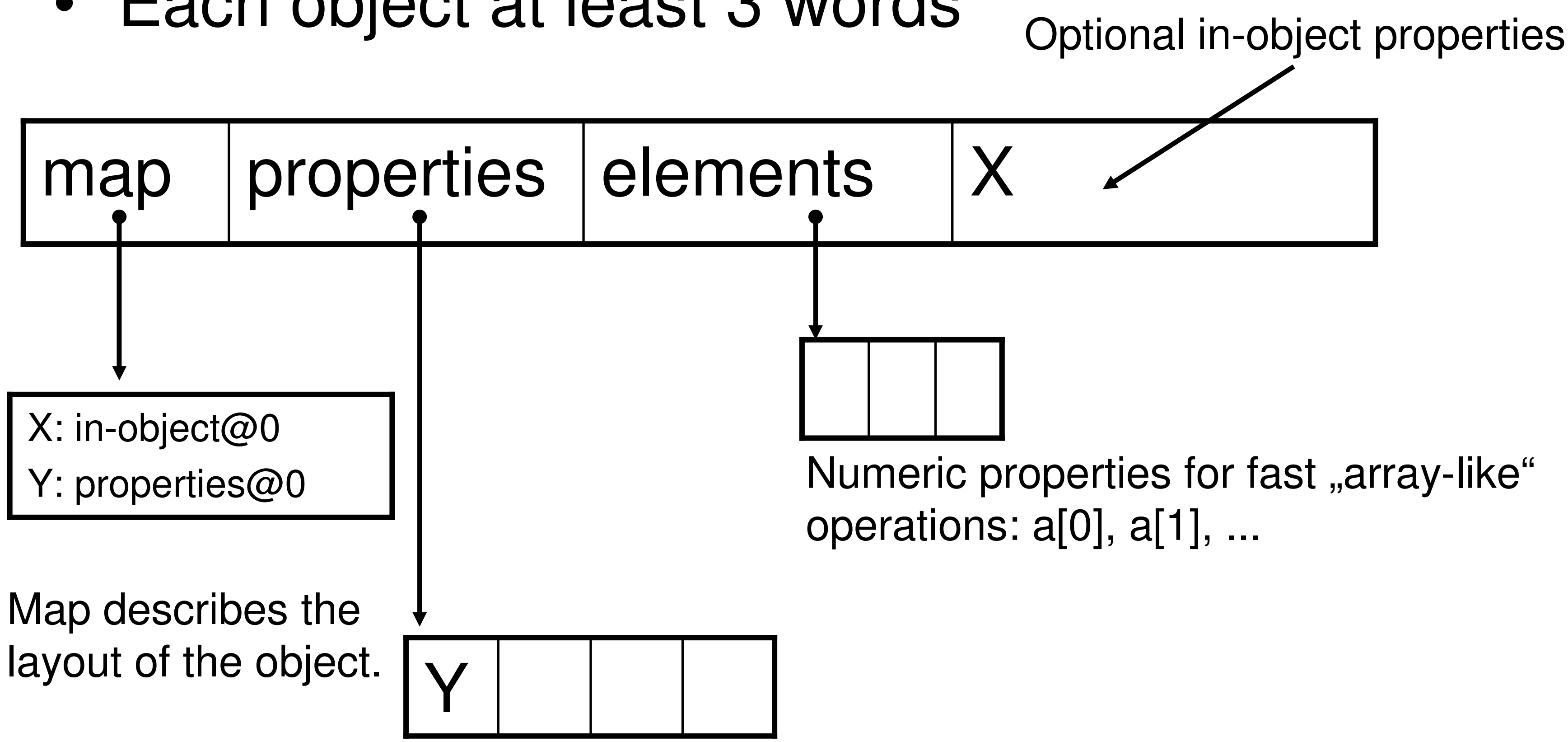

```
var o = {};
o.a = 0;
o.b = 0;
```

Maps summary

- Object shape stable after construction
  - Allows classical class-based ICs
- Map checks instead of class checks
  - Optimized by redundancy elimination
  - Calls may change an objects map
- Cost of changeable object shape

V8 object layout

- Each object at least 3 words

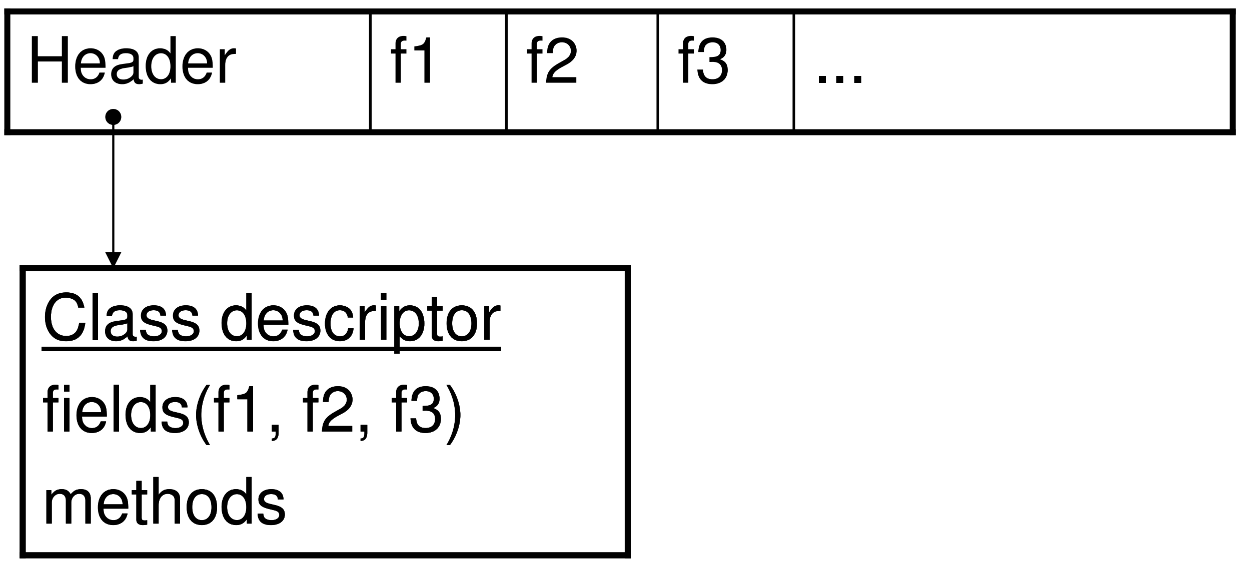


V8 object layout continued

- Object can be fast or slow mode
  - Properties/elements array
  - Hash-table based dictionary
- Different representations for elements array
  - With or without holes
  - Double-only, int-only
- Allocate right amount for in-object properties
  - In-object slack tracking

Dart VM objects

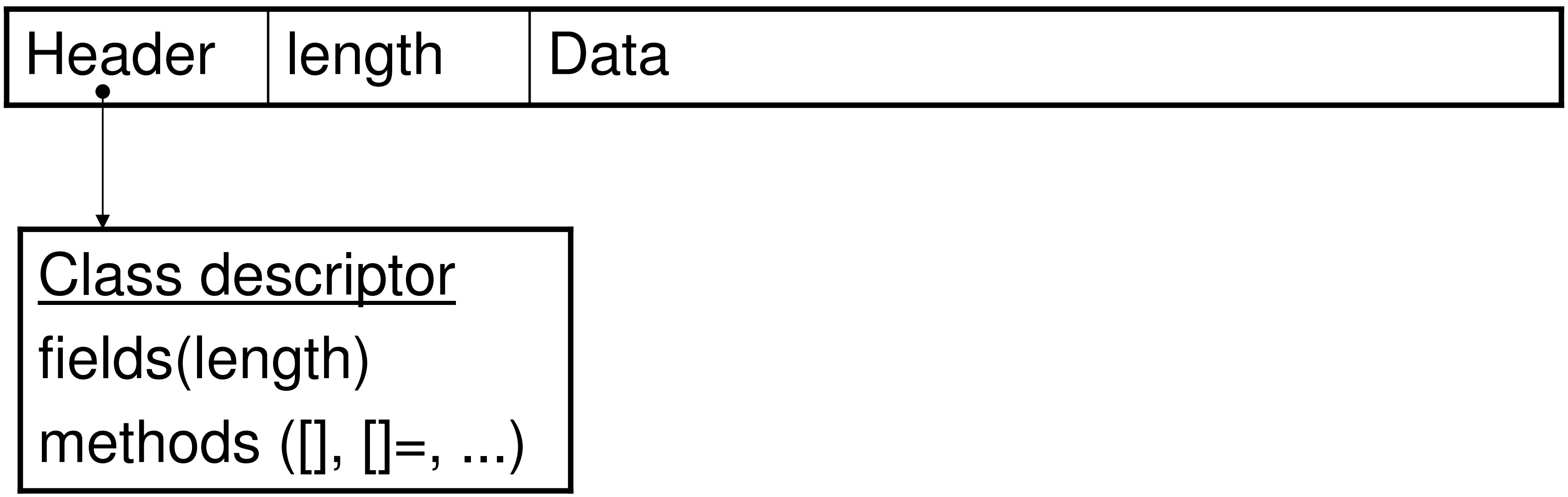
- 1 word header



- Instances have known length

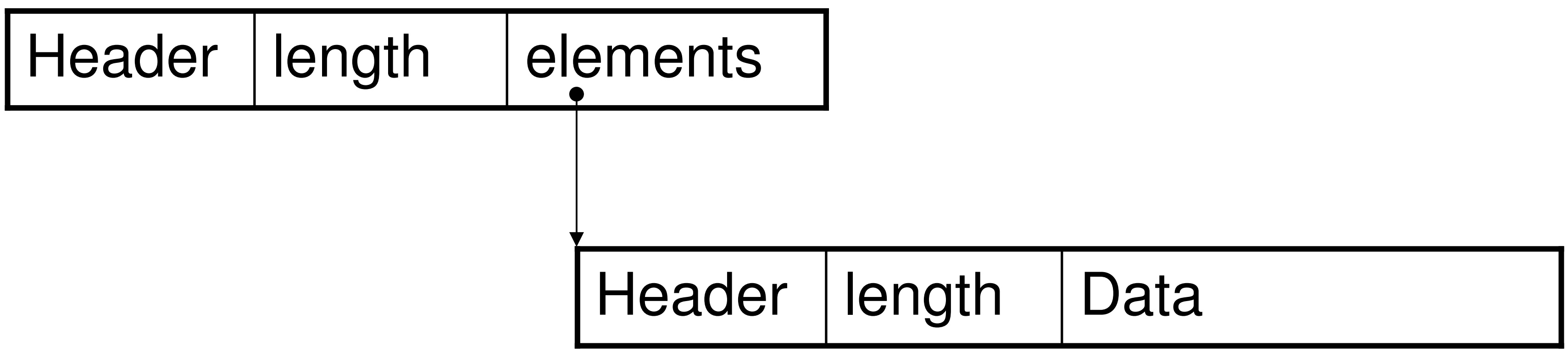
Dart VM arrays

- Elements stored in-line



Dart VM growable arrays

- Elements stored in separate fixed array



Array example JS

```
function sum(a) {
  var r = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] == null) {
      f(a, i);
    }
  }
}
```

- May change representation of `a` (elements dense/sparse/smi-only)
- May change length of `a`
  - JS arrays are always variable-length

Array load example

- Operations for a[i]
  1. `Check class/map`
  2. `Check bounds`
  3. `Load elements`
  4. `Load data`

Array example Dart

```
void sum(List a) {
  var r = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] == null) {
      f(a, i);
    }
  }
}
```

- f may change length of a, only if a is a growable
- a[i] => 3+1 operations (+1 for growable list)
- Allows hoisting class check before of the loop
  - e.g. using PRE
  - Bounds check too, if array is fixed-length

Optimizing compiler

- Problem: Most operations method calls
    - Even arithmetic ops
  - Defeats common optimizations
    - Register allocation
    - Redundancy elimination
    - etc.
- Idea: Generate a version without calls
  - Inline common case, but:
  - What is the common case?

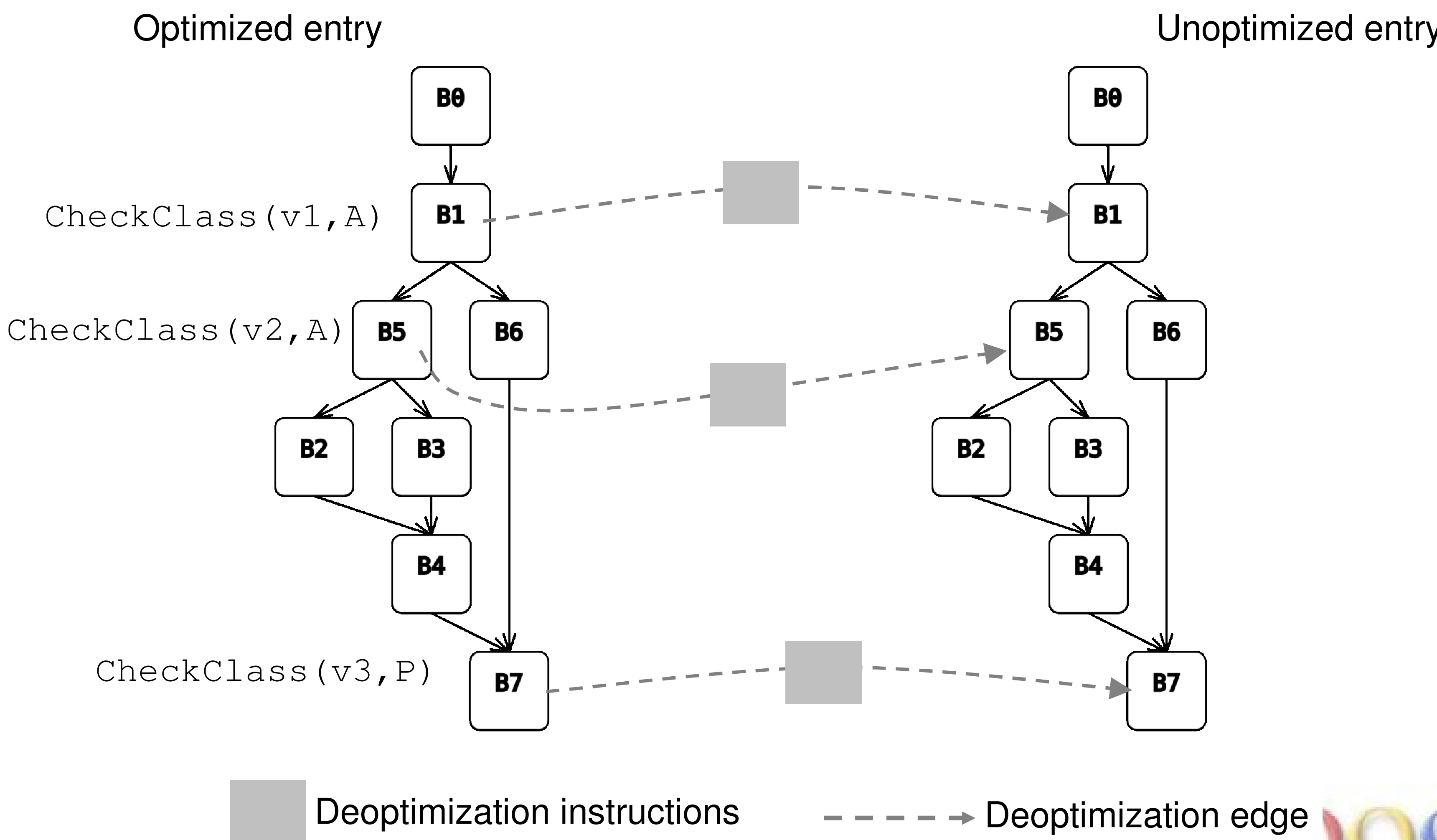
IC data as run-time feedback

- Idea: Separate compiled bodies for cache-hit and cache-miss case
- Mine ICs for types encountered so far
- Two-stage compilation:
  1. Generic version (handles all cases)
  2. Specialized version (only common cases)

Compilation strategy

- Run first compiled version
  - ICs collect types encountered
- After some time (runtime profiler)
  - Compile a new optimized version
  - IC arrays used for type-specialized code
  - Translate to SSA form
- Speculate that types seen so far are stable
  - Insert checks
  - Keep first version as a fall-back

Optimized flow graph



Deoptimization

1. Enter C++
2. Execute deoptimization instructions
3. Re-enter Dart

- Optimized state → unoptimized state
- State = Registers + Stack
- Compiled with optimized code
  - Byte-code interpreted by C++ routine
- Also needed debugging, stack traces

Deoptimization summary

- "Decouple" fast from slow path
  - Profitable data-flow analysis
  - Optimized code unaffected by slow path
- Compiler optimizations work well
  - Optimized code has fewer calls
  - Deoptimization checks assert types
  - Many redundant checks can be eliminated

Summary

- Programmers often just want classes
  - Popular libraries to emulate classes in JS
- Language design impacts performance
  - Cost from language semantics
  - Cost from complexity in implementation
- Goal with Dart:
  - High and predictable performance

Links

Dart

http://dartlang.org

http://dartlang.org/performance

http://code.google.com/p/dart

V8

http://code.google.com/p/v8

- Both open source under BSD-style license