JavaScript as a compilation target
Making it fast

Florian Loitsch, Google

Who am I?

Florian Loitsch, software engineer at Google

Projects

- **Scheme2Js** - Scheme-to-JavaScript compiler
- **Js2scheme** - JavaScript-to-Scheme compiler
- **V8** - high-performance JavaScript virtual machine
- **Dart** - structured programming for the web

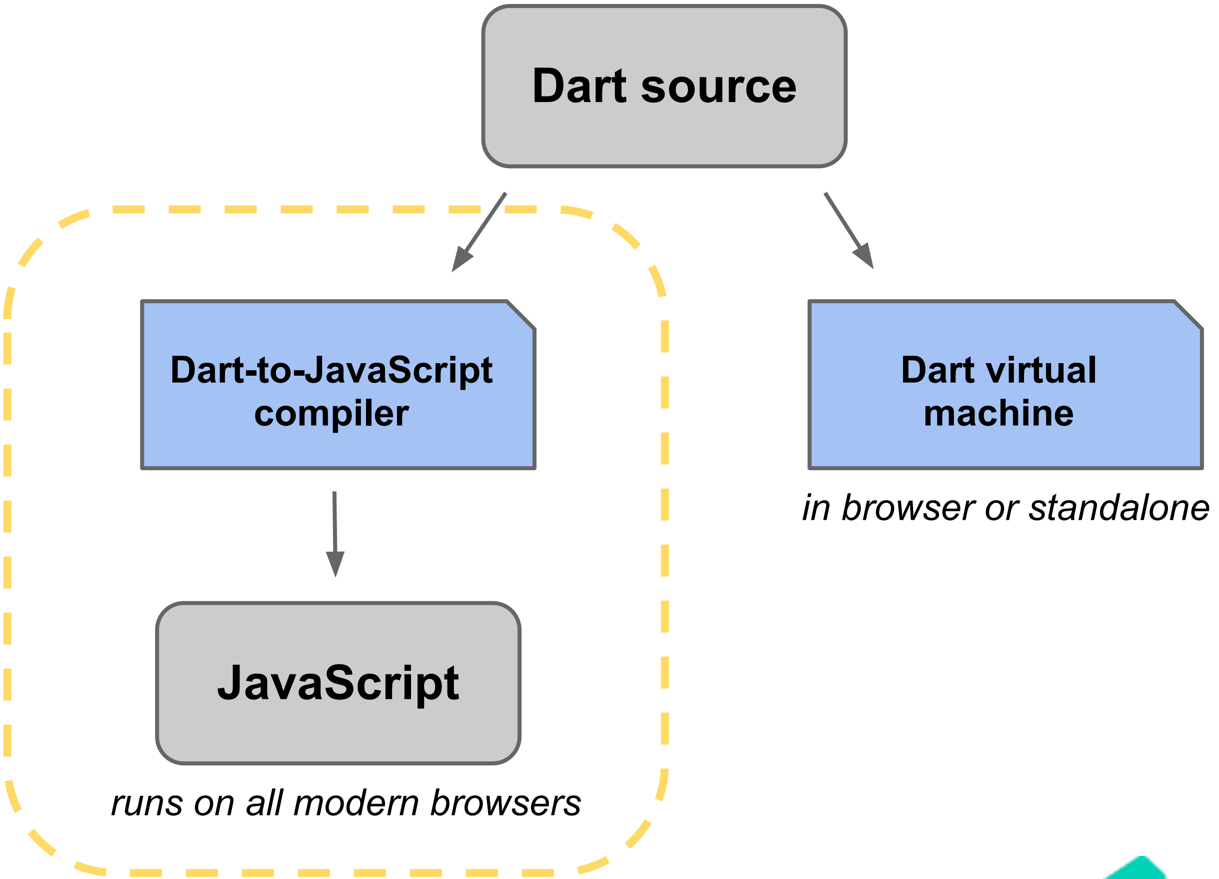
What is Dart?

- Unsurprising object-oriented programming language
- Class-based single inheritance
- Familiar syntax with proper lexical scoping
- Optional static type annotations

```
main() {
  for (int i = 99; i > 0; i--) {
    print("$i bottles of beer on the wall, ....");
    print("Take one down and pass it around ...");
  }
}
```

DART

Dart execution and deployment



Dart-to-JavaScript compiler goals

- Support Dart apps on all modern browsers
  - Tested on Chrome, Firefox, IE, and Safari
  - Ensures that the use of the Dart VM is optional

- Generate efficient and compact JavaScript

Example: What's the point?

Source code in Dart

```
main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  var distance = p.distanceTo(q);
  ...
}
```

Example: What's the point?

Compiled JavaScript code

```
$.main = function() {
  var p = $.Point(2, 3);
  var q = $.Point(3, 4);
  var distance = p.distanceTo$1(q);
  ...
};
```

```
main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  var distance = p.distanceTo(q);
  ...
}
```

DART

Example: What's the point?

- Static functions are put on the `$` object
  - Top-level functions such as `$.main`
  - Factory functions such as `$.Point`

- Method calls are translated to functions calls
  - Arity is encoded in the selector (`distanceTo$1`)
  - Supports named optional arguments

Language challenges

Closures

- JavaScript doesn't check arity
- Encoding the arity into the name does not work for closures

- Dart supports named arguments
- Dart `call` operator makes objects behave like closures

Operators

- JavaScript implicitly converts + inputs to numbers or strings

```
1 + "2";  // => "12"
1 - "2";  // => -1
{} + 3;   // => 3
3 * {};   // => NaN
```

- Dart has user-defined operators

Array accesses

- JavaScript has no notion of out of bounds access and all keys are treated as strings

- In JavaScript, the index operator works on every object

```
var a = [ 1, 2 ];
a[2];    // => undefined.
a[1.5];  // => undefined
a["1"];  // => 2
var t = {};
t[0];    // => undefined.
```

Solutions

Closures - Solution

- Treat closures as class instances
  - Use instance fields for captured (boxed) variables
  - Use methods for implementing calling conventions

- Allocating small JavaScript objects is fast!
  - New JavaScript closure ~ new object with six fields

Example: Closures

Source code in Dart

```
main() {
  var list = [ 1, 2, 3 ];
  print(list.map((each) => list.indexOf(each)));
}
```

DART

Example: Closures

Compiled JavaScript code

```
$.main = function() {
  var list = [1, 2, 3];
  $.print($.map(list, new $.main$closure(list)));
};

$.main$closure = {"": ["list"],
  call$1: function(each) {
    return $.indexOf$1(this.list, each);
  }
};
```

Note the compact class representation

```
main() {
  var list = [ 1, 2, 3 ];
  print(list.map((each) => list.indexOf(each)));
}
```

Operators - Naive Solution

- Implement operator methods.

```
Number.prototype.add = function(x) {
  if (!isNumber(x)) throw $.ArgumentError(x);
  return this + x;
};
Array.prototype.index = function(i) {
  if (!isInteger(i)) throw $.ArgumentError(i);
  if (i < 0 || i > this.length)
      throw $.RangeError(i);
  return this[i];
};
```

Operators - Naive Solution

- Implement operator methods.

```
Number.prototype.add = function(x) {
  if (!isNumber(x)) throw $.ArgumentError(x);
  return this + x;
};
Array.prototype.index = function(i) {
  if (!isInteger(i)) throw $.ArgumentError(i);
  if (i < 0 || i > this.length)
      throw $.RangeError(i);
  return this[i];
};
```

```
x + y;
list[i];
```

$\Rightarrow$

```
x.add(y);
list.index(i);
```

Operators - Naive Solution
Drawbacks

- Pollutes global objects (`Number`, `Array`, `String`, `Object` ...)
- Can be very slow. (Especially without strict mode)

Interceptors

- Static interceptors

```
typeof y === 'number'
```

```
$.add = function(x, y) {
  if (isNumber(x) && isNumber(y)) return x + y;
  if (isNumber(x)) throw $.ArgumentError(y);
  if (isDartObject(x)) return x.add$1(y);
  throw noSuchMethodException(x, "+", y);
};
$.index = function(a, i) {
  if (isArray(a)) {
    if (!isInteger(i)) throw $.ArgumentError(i);
    if (i < 0 || i > a.length) throw $.RangeError(i);
    return a[i];
  }
  ...
};
```

Example: Sum the elements of a list

Source code in Dart

```
main() {
  var list = [ 2, 3, 5, 7 ];
  var sum = 0;
  for (var i = 0; i < list.length; i++) {
    sum += list[i];
  }
  print("sum = $sum");
}
```

Example: Sum the elements of a list

Compiled JavaScript code

```
$.main = function() {
  var list = [2, 3, 5, 7];
  for (var sum = 0, i = 0; $.ltB(i, $.length(list)); i = $.add(i, 1)) {
    sum = $.add(sum, $.index(list, i));
  }
  $.print('sum = ' + $.S(sum));
}
```

```
main() {
  var list = [ 2, 3, 5, 7 ];
  var sum = 0;
  for (var i = 0; i < list.length; i++) {
    sum += list[i];
  }
  print("sum = $sum");
}
```

DART

Performance
on V8

Slow:

- about ~6 times slower than handwritten JavaScript

- for some benchmarks up to ~23 times slower

Types

First improvement:

- Avoid global interceptors when the type is known

- Track types and do (local) type-inference

Example: Sum the elements of a list

Compiled JavaScript code with local type inference

```
$.main = function() {
  var list = [1, 2, 3, 4];
  for (var sum = 0, i = 0; i < list.length; ++i) {
    // Check that the index is within range before
    // reading from the list.
    if (i < 0 || i >= t1) throw $.ioore(i);
    var t1 = list[i];
    // Check that the element read from the list is
    // a number so it is safe to use + on it.
    if (typeof t1 !== 'number') throw $.iae(t1);
    sum += t1;
  }
  $.print('sum = ' + $.S(sum));
};
```

Performance
Type inference

much better:

- 50%+ speed of handwritten JS code

- Global type inference algorithm not good enough:
  - Still too many calls to the interceptors
  - We don't know if the instructions have side-effects

Speculative Optimizations

Speculative Optimizations

- Try to guess the type of an instruction based on its inputs and uses
- Optimize for the speculated type
- Bailouts in case the guess was wrong

Speculative optimizations (1)

It would be great if `x` was a JavaScript array

```
sum(x) {
  var result = 0;
  for (var i = 0; i < x.length; i++) {
    result += x[i];
  }
  return result;
}
```

Speculative optimizations (2)

We really hope x is a JavaScript array

```
$.sum = function(x) {
  if (!$.isJsArray(x)) return $.sum$bailout(1, x);
  var result = 0;
  for (var t1 = x.length, i = 0; i < t1; ++i) {
    if (i < 0 || i >= t1) throw $.ioore(i);
    var t2 = x[i];
    if (typeof t2 !== 'number') throw $.iae(t2);
    result += t2;
  }
  return result;
};
```

Speculative optimizations (3)

What if it turns out `x` is not a JavaScript array?

```
$.sum$bailout = function(state, x) {
  var result = 0;
  for (var i = 0; $.ltB(i, $.get$length(x)); ++i) {
    var t1 = $.index(x, i);
    if (typeof t1 !== 'number') throw $.iae(t1);
    result += t1;
  }
  return result;
};
```

Heuristics for speculating

- To avoid generating too much code we need to control the speculative optimizations
- Hard to strike the right balance between optimizing too little and too much
- **Current solution:** Only speculate about types for values that are used from within loops

Profile guided optimizations

What if we aggressively speculated about types and used profiling to figure out if it was helpful?

1. Use speculative optimizations everywhere!
2. Profile the resulting code
3. Re-compile with less speculation

Don't keep optimized methods that are rarely used or always bail out

DART

Performance
Speculative Optimizations

Really pays off:

- ~75% speed of handwritten code

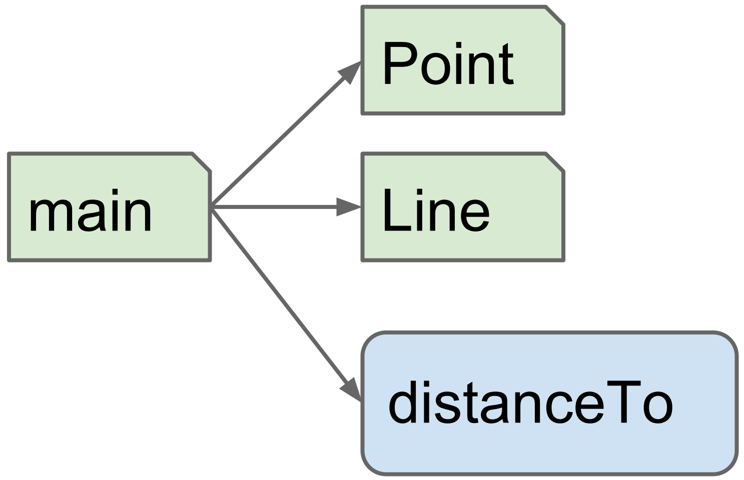
Space/Speed Trade-off:

- 15% code increase when using the heuristics
- 72% code increase when always speculating

Tree shaking

Tree shaking

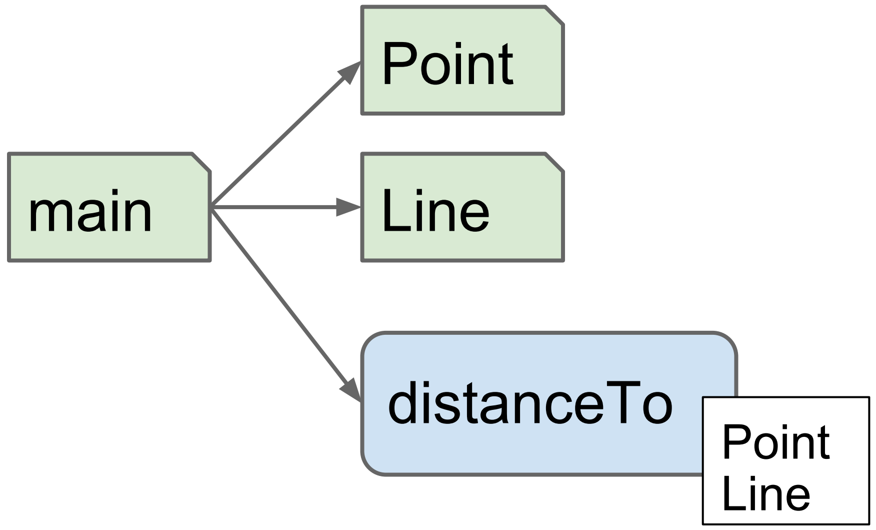
- Only compile functions that are potentially reachable.



```
main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  var distance = p.distanceTo(q);
  var line = new Line(p, q);
  ...
}
```

Tree shaking

- Only compile functions that are potentially reachable.

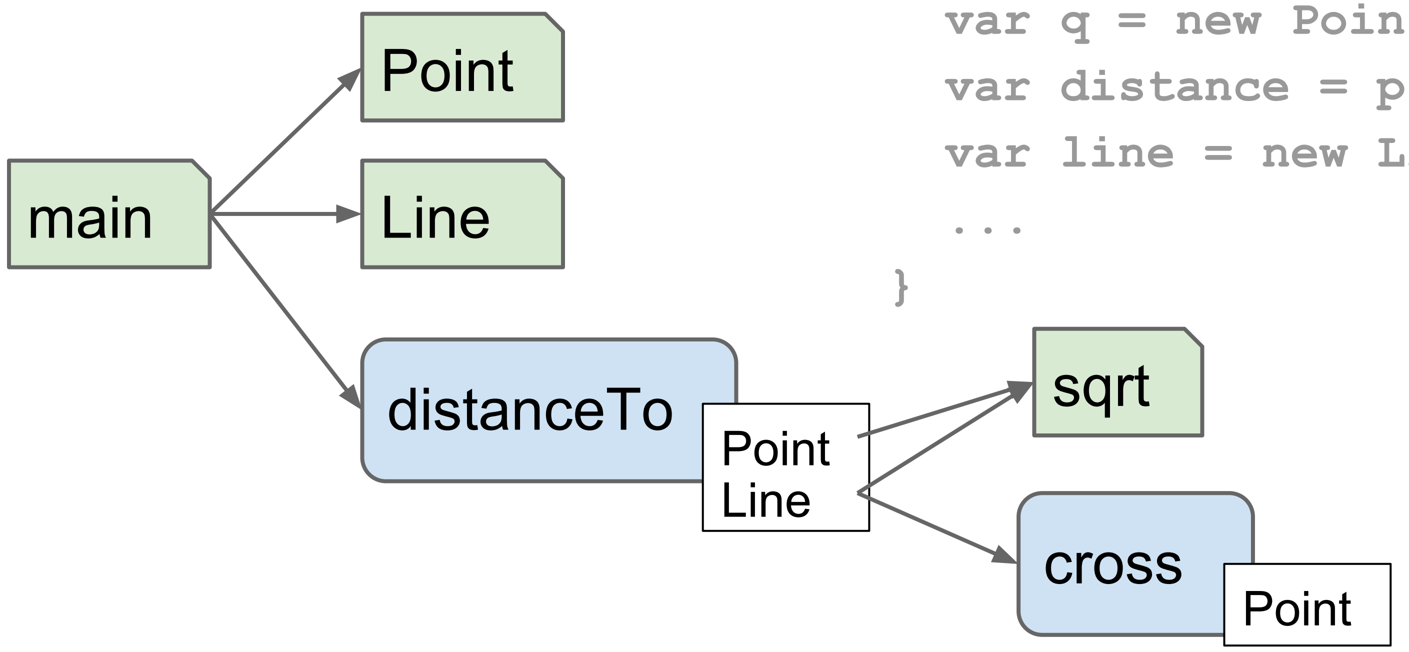


```
main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  var distance = p.distanceTo(q);
  var line = new Line(p, q);
  ...
}
```

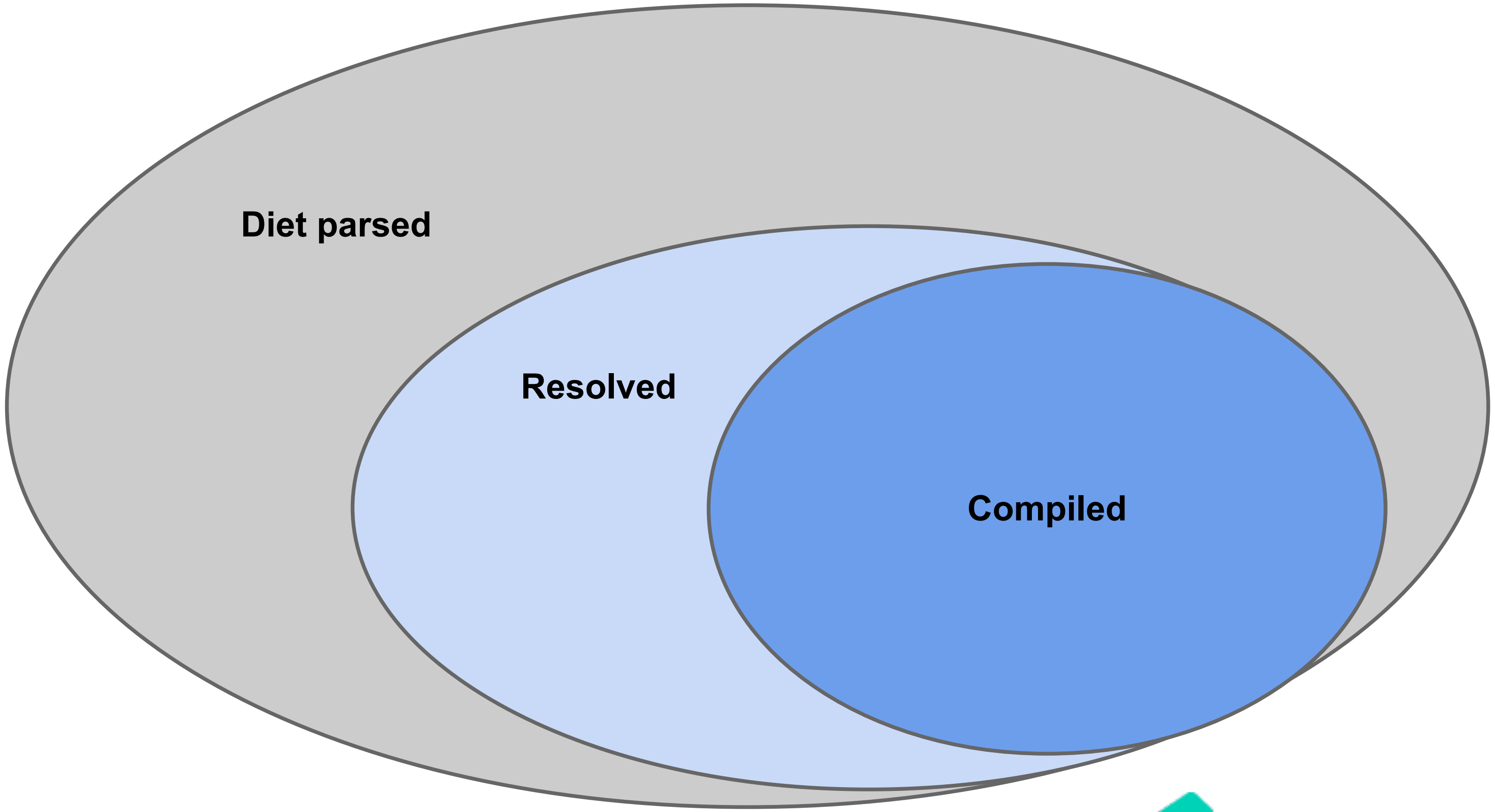
Tree shaking

- Only compile functions that are potentially reachable.

```
main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  var distance = p.distanceTo(q);
  var line = new Line(p, q);
  ...
}
```

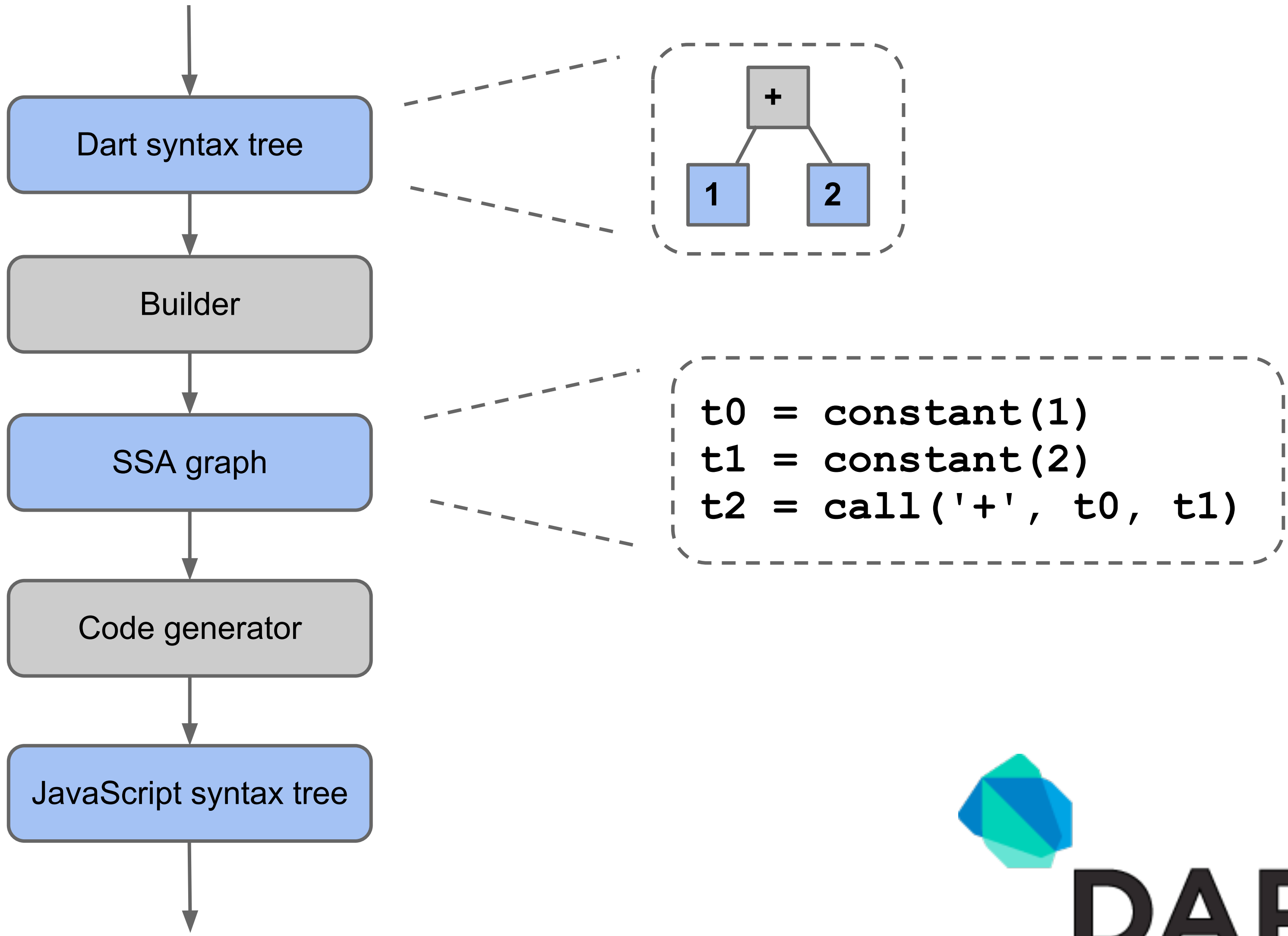


Code after tree shaking



SSA

Intermediate Representations



```
t0 = constant(1)
t1 = constant(2)
t2 = call('+', t0, t1)
```

SSA - Single Static Assignment

SSA form convenient for analyses and optimizations:

- Local type inference
- Function inlining
- Global value numbering
- Loop-invariant code motion
- Range propagation

DART

Global Value Numbering

GVN: replace instructions with equal ones from dominators if no side-effects can affect the outcome.

```
if (compiler.generated != null) {
  return compiler.generated;
}
```

```
var t0 = compiler.generated;
if (t0 != null) {
  return t0;
}
```

DART

Loop invariant code motion

LICM: hoist instructions out of loops if its computation doesn't have any side-effect, and the value is independent of the loop's body.

```
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```

$$\Downarrow$$

```
var t0 = a.length;
for (int i = 0; i < t0; i++) {
  sum += a[i];
}
```

Example: Sum the elements of a list

Compiled JavaScript code with all optimizations

```
$.main = function() {
  var list = [1, 2, 3, 4];
  for (var t1 = list.length, sum = 0, i = 0; i < t1; ++i) {
    var t2 = list[i];
    // Check that the element read from the list is
    // a number so it is safe to use + on it.
    if (typeof t2 !== 'number') throw $.iae(t2);
    sum += t2;
  }
  $.print('sum = ' + $.S(sum));
};
```

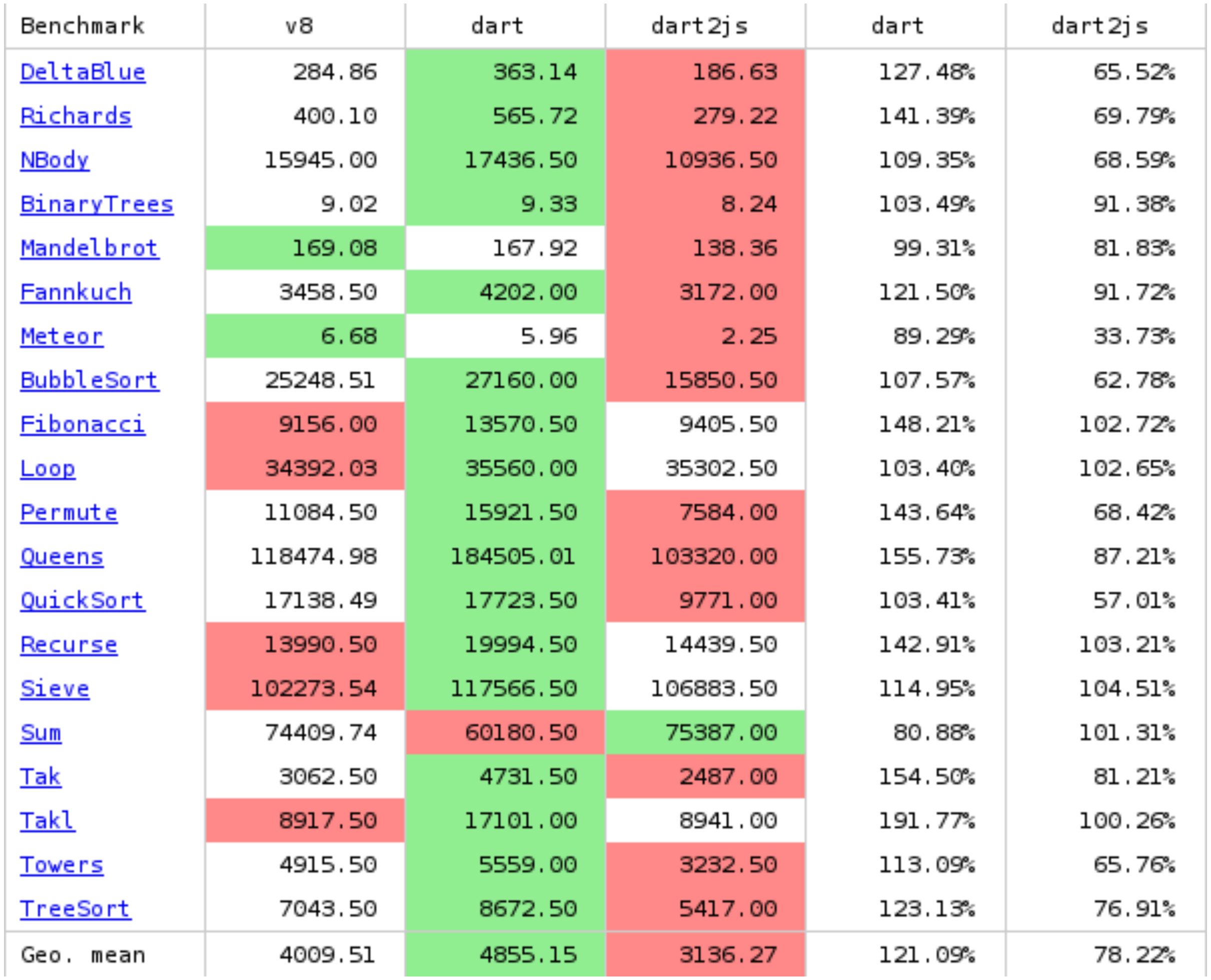
DART

Status

Code size

- Size of the generated code has improved since our first release!
- If your app translates to sizeable chunks of JavaScript it could be because of imports
- Work on supporting minification is in progress (use --minify option)

Performance



Benchmark	v8	dart	dart2js	dart	dart2js
DeltaBlue	284.86	363.14	186.63	127.48%	65.52%
Richards	400.10	565.72	279.22	141.39%	69.79%
NBody	15945.00	17436.50	10936.50	109.35%	68.59%
BinaryTrees	9.02	9.33	8.24	103.49%	91.38%
Mandelbrot	169.08	167.92	138.36	99.31%	81.83%
Fannkuch	3458.50	4202.00	3172.00	121.50%	91.72%
Meteor	6.68	5.96	2.25	89.29%	33.73%
BubbleSort	25248.51	27160.00	15850.50	107.57%	62.78%
Fibonacci	9156.00	13570.50	9405.50	148.21%	102.72%
Loop	34392.03	35560.00	35302.50	103.40%	102.65%
Permute	11084.50	15921.50	7584.00	143.64%	68.42%
Queens	118474.98	184505.01	103320.00	155.73%	87.21%
QuickSort	17138.49	17723.50	9771.00	103.41%	57.01%
Recurse	13990.50	19994.50	14439.50	142.91%	103.21%
Sieve	102273.54	117566.50	106883.50	114.95%	104.51%
Sum	74409.74	60180.50	75387.00	80.88%	101.31%
Tak	3062.50	4731.50	2487.00	154.50%	81.21%
Takl	8917.50	17101.00	8941.00	191.77%	100.26%
Towers	4915.50	5559.00	3232.50	113.09%	65.76%
TreeSort	7043.50	8672.50	5417.00	123.13%	76.91%
Geo. mean	4009.51	4855.15	3136.27	121.09%	78.22%

DART

Conclusions

- Speculative optimizations are crucial for speed

- Compiled Dart now at 75%+ speed of handwritten JavaScript code

- You should give Dart a try

DART

Questions?

I will be at the Google booth during the next coffee break for more Q&A.

DART