Compiling Dart to JavaScript

Karl Klose, Software Engineer at Google

Dart: An Overview

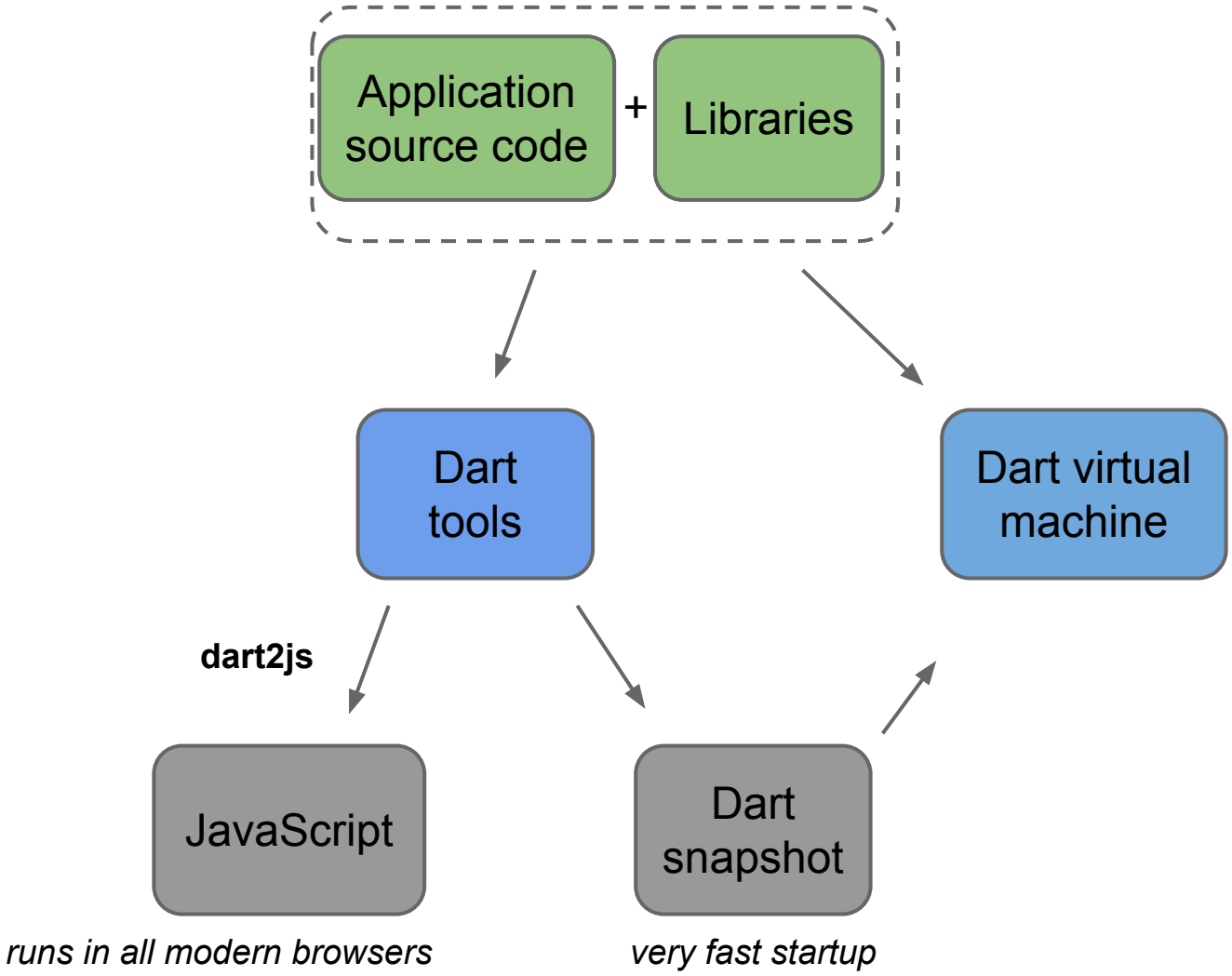
The Dart Platform

- Language
- Libraries
- IDE support
- Virtual machine
- Compiles to JavaScript

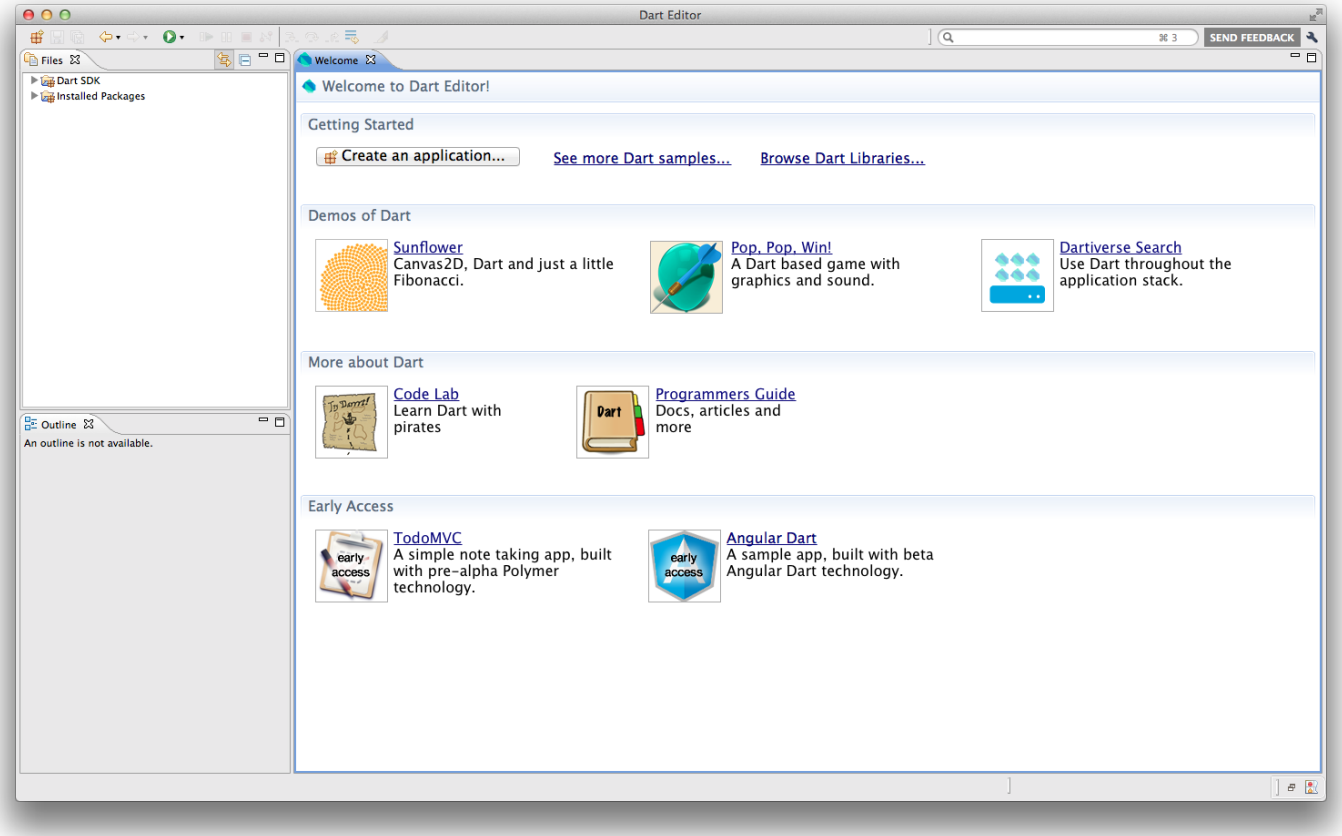
State of the Project

- Dart 1.0 shipped in November 2013
- Dart is ECMA Standard (TC 52)
- Current Version: 1.8

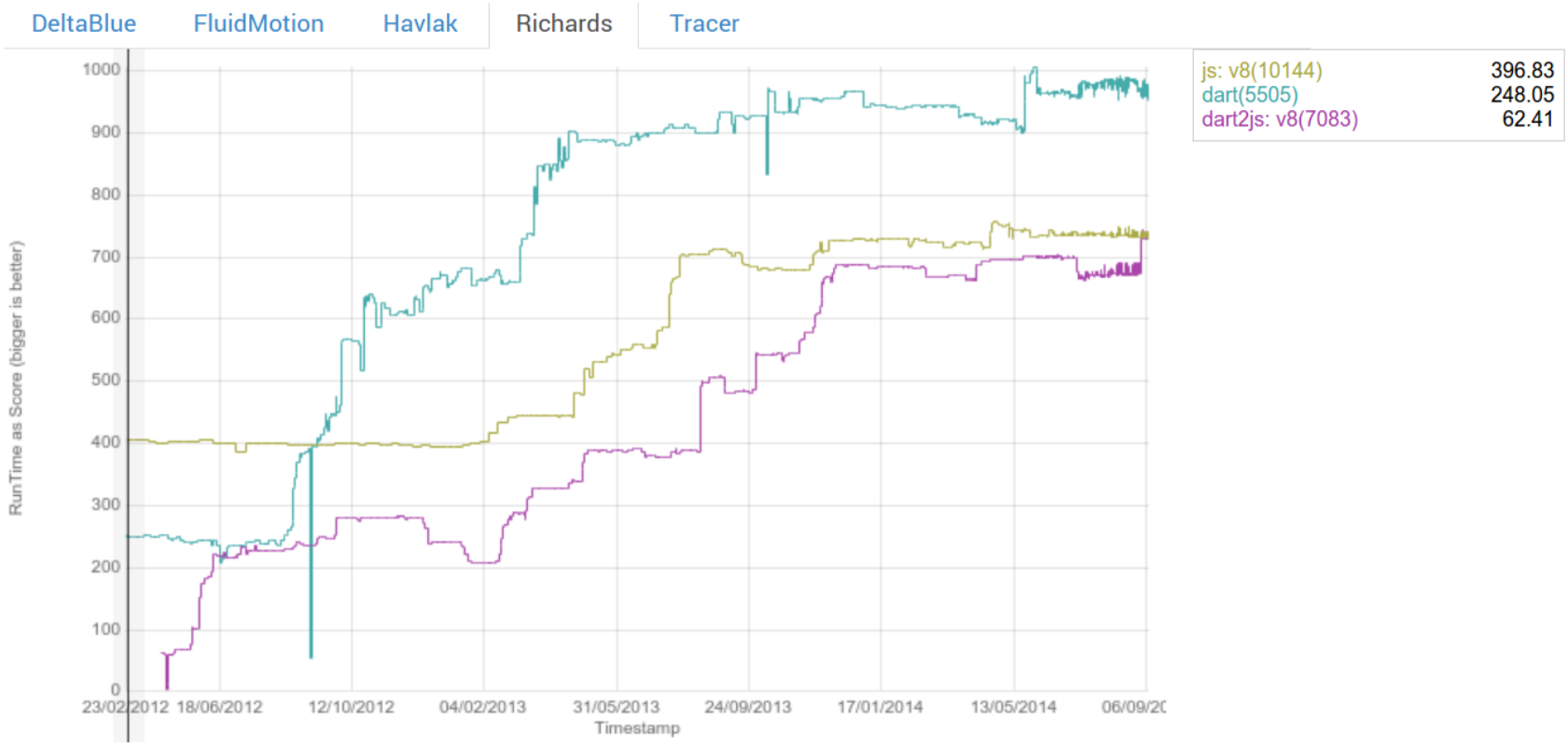
Application Deployment



Tool Support: Analyzer, Dart Editor, IntelliJ, ...



Performance



Language Features

The Language

- Familiar
- Unsurprising semantics
- Static program structure
- Optional typing
- Single threaded, isolates

Functions

```
square(int i) => i * i;

inc(i, [by = 1]) => i + by;
dec(i, {by: 1}) => i - by;

inc(1, 2);
dec(4, by: 2);
```

Functions

```
twice(f(x)) => (v) => f(f(v));


var v = 0;
var f = (x) => x + v;
v = 1;
print(f(2));


var l = [1, 4, 7, 2];
var r = [];
for (int i = 0; i < l.length; i++) {
  if (l[i] > 3) r.add(() => i);
}
```

Classes

```
class Point {
  var x;
  var y;
  Point(this.x, this.y);
  toString() => "($x, $y)";
}

main() {
  print(new Point(1, 2));
  print(new Point(1.5, 2.0));
}
```

Classes: Constructors

```
class A {
  var f;
  A() : f = 1;
  A.withValue(this.f);
  A.defaultValue() : this.withValue(42);
  factory A.fact(v) => new A.withValue(v);
  factory A.redirect() = B;
}

class B extends A {
  B() : super.withValue(27);
}
```

Classes: Subtypes and Mixins

```
class M {
  foo() {}
}

class A {
  bar() {}
}

class B extends A implements M {
  foo() {}
}

class C extends A with M {}
class D = A with M;
```

Classes: Generic Types

```
class Point<T> {
  T x;
  T y;
  Point(this.x, this.y);
  toString() => "($x, $y)";
}

main() {
  print(new Point<int>(1, 2));
  print(new Point<double>(1.5, 2.0));
}
```

Classes: Operators

```
class Point {
  var x;
  var y;
  Point(this.x, this.y);
  operator +(Point other) {
    return new Point(x + other.x, y + other.y);
  }
}

main() {
  print(new Point(1, 2) + new Point(2, -5));
}
```

Classes: The `call` Operator

```
class Fun implements Function {
  int call(int i) => i + 1;
}

main() {
  var f = new Fun();
  print([1, 2, 3].map(f));
}
```

Iterators

```
class Collection {
  List storage = [1, 2, 3];
  get iterator => storage.iterator;
}

main() {
  for (var i in new Collection()) {
    print(i);
  }
  new Collection().forEach(print);
}
```

Method Cascades

```
class ContactBuilder {
  void firstName(String s) { /* ... */ }
  void lastName(String s) { /* ... */ }
  Contact done();
}

main() {
  new ContactBuilder()
    ..firstName('Jon')
    ..lastName('Doe')
    .done();
}
```

Getters and Setters

```
class C {
  int _foo;
  get foo => _foo == null ? 0 : _foo;
  set foo(value) {
    _foo = value;
  }
}

main() {
  var c = new C();
  print(c.foo += 42);
}
```

Type Tests, Casts and Type Promotion

```
class A { a() {} }

class B extends A { b() {} }

main() {
  A value = new B();
  value.b();  // warning
  print(value is B);
  (value as B).b();
  (new A() as B).b();  // throws!
  print(value is B && value.b());
}
```

Warnings and Errors

- Static warning
  - Emitted by static analyzer, dart2js
  - No effect on runtime system
- Compile-time error
  - Emitted before running a piece of code
  - May be very late, e.g., in the VM
  - Stops current isolate
- Runtime error
  - Thrown on illegal program execution
  - Can be caught and recovered from

The Static Type System

- Type annotations are optional
- Normally not checked at runtime
  - Checked mode checks type
  - Production mode ignores types
- Static type warning, if types are unrelated
- Type arguments are preserved at runtime

Compilation to JavaScript

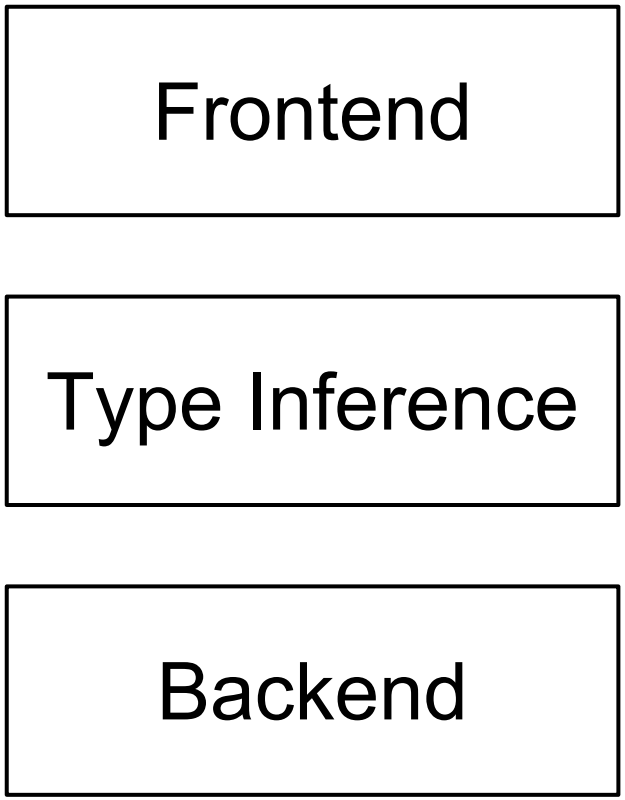
Compilation To JavaScript

- JavaScript as Compilation Target
- Structure of the Compiler
- Emitting efficient and small code

JavaScript as Compilation Target

- No assembly language for the web
  - too high-level
  - performance is not easy to predict
  - performance depends on platform (browser)
- Big semantic gap between Dart and JavaScript

Structure of the Compiler

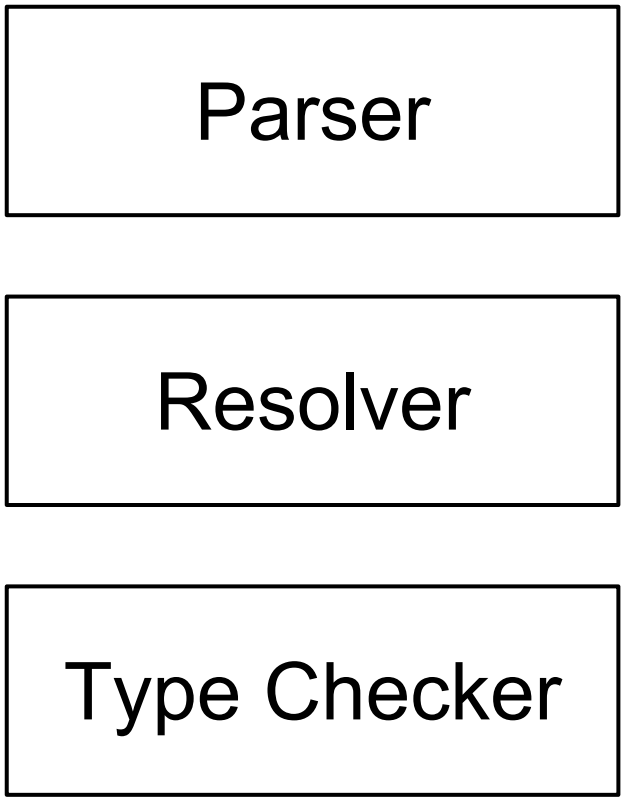


Create a semantic model of the program.

Globally, sound types analysis.

Generate and optimize intermediate representation and emit JavaScript.

Structure of the Compiler: Frontend

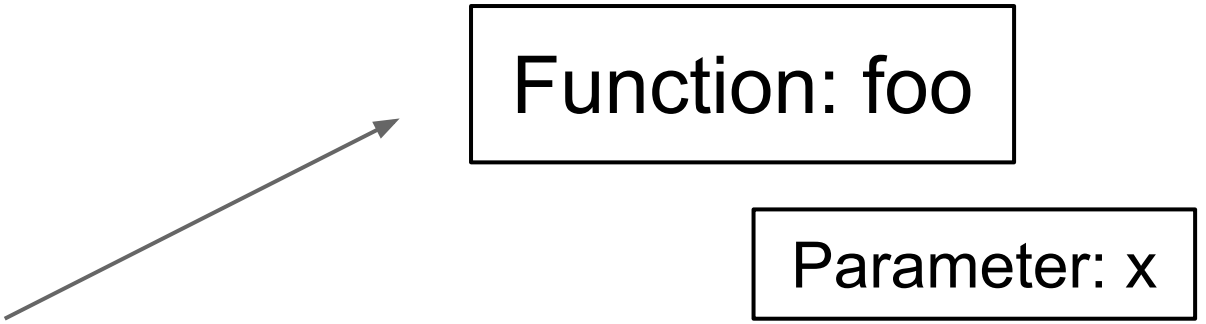


Creates AST nodes and structural entities (elements).

Resolves identifiers to the element or type they refer to. Performs semantic checks.

Checks the program against violations of the static type system rules.

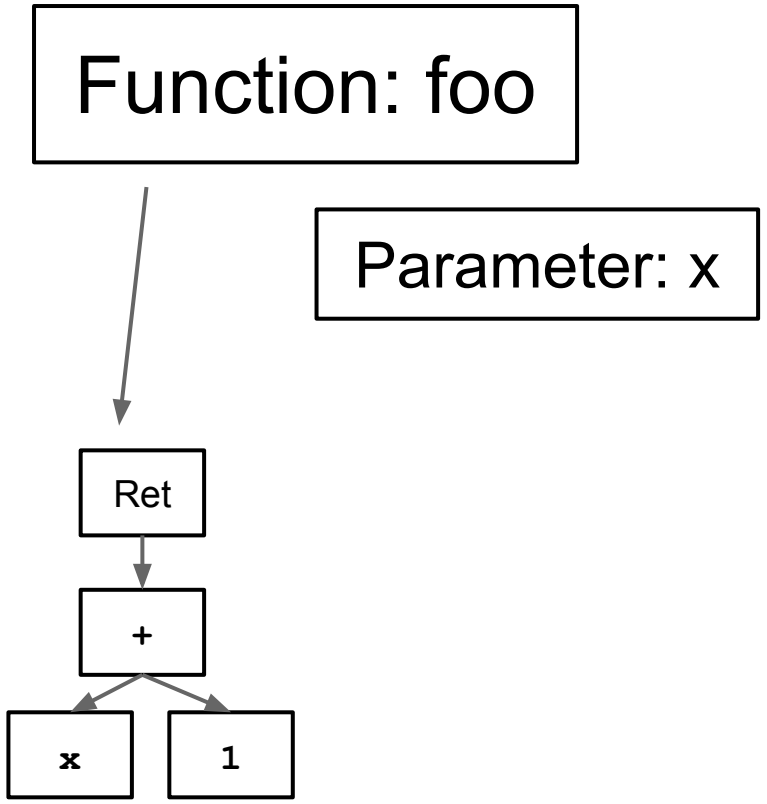
Partial (Diet-) Parsing




```
int foo(var x) {
  return x + 1;
}
```

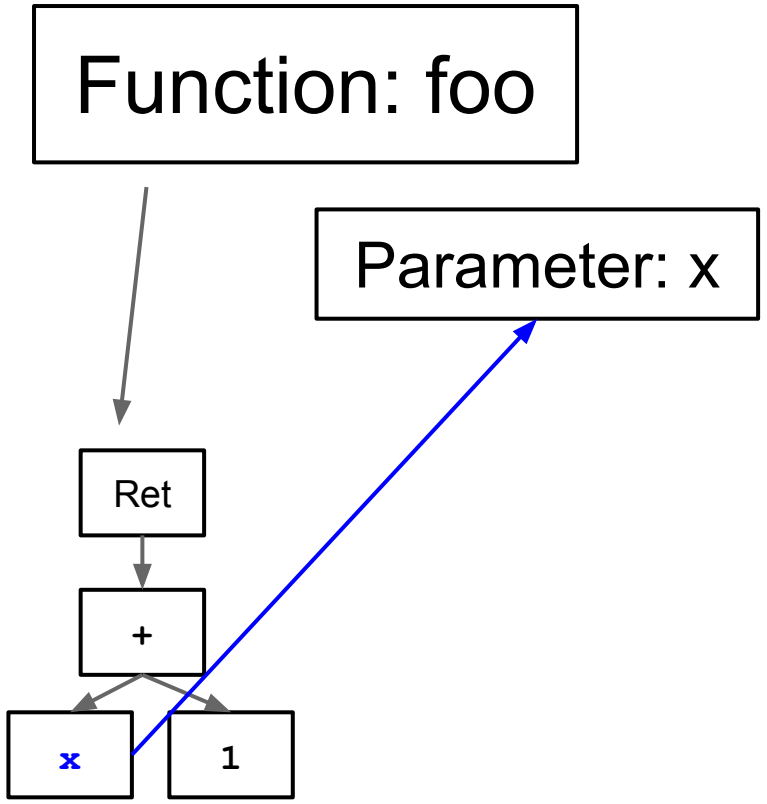
Full Parsing

```
int foo(var x) {
  return x + 1;
}
```



Resolution

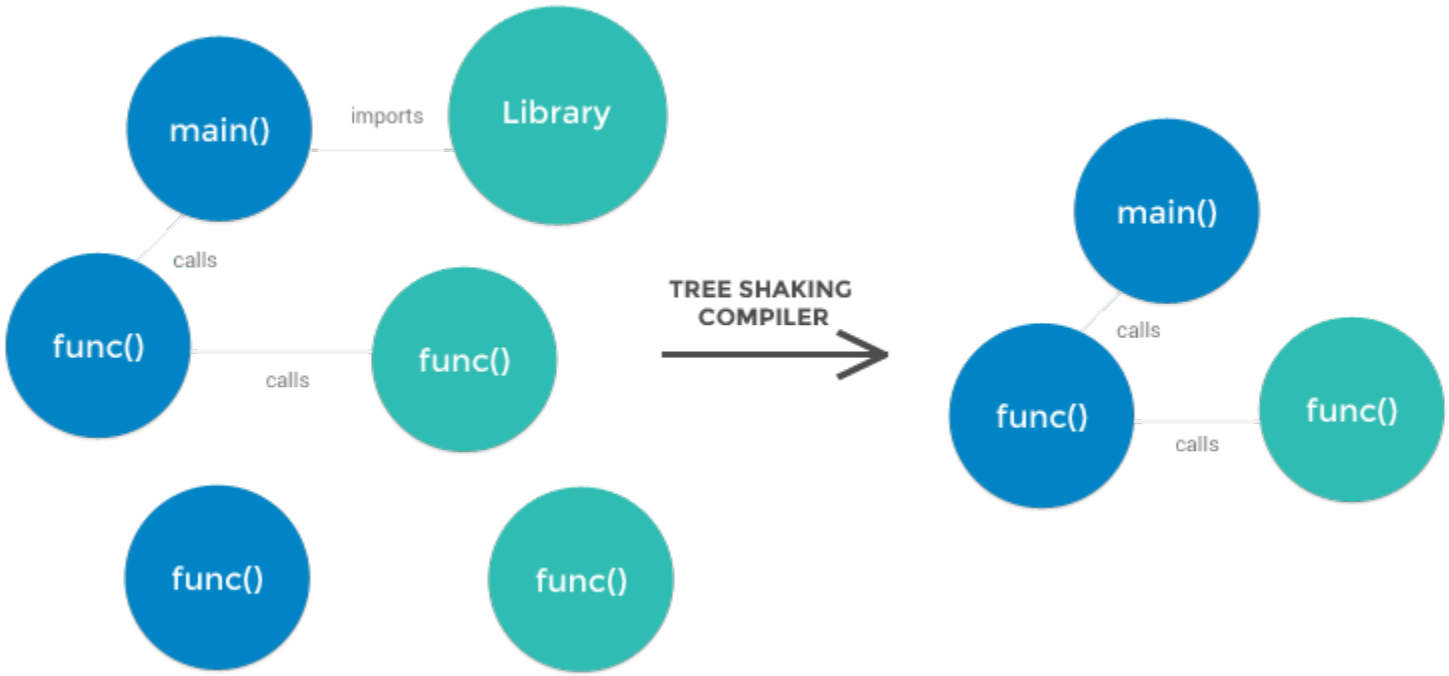
```
int foo(var x) {
  return x + 1;
}
```



Resolution

- Works on one function at a time
- Resolves identifiers to elements or types
- Resolve and validate class hierarchies of referenced types
- Triggers resolution of possible targets (tree shaking)

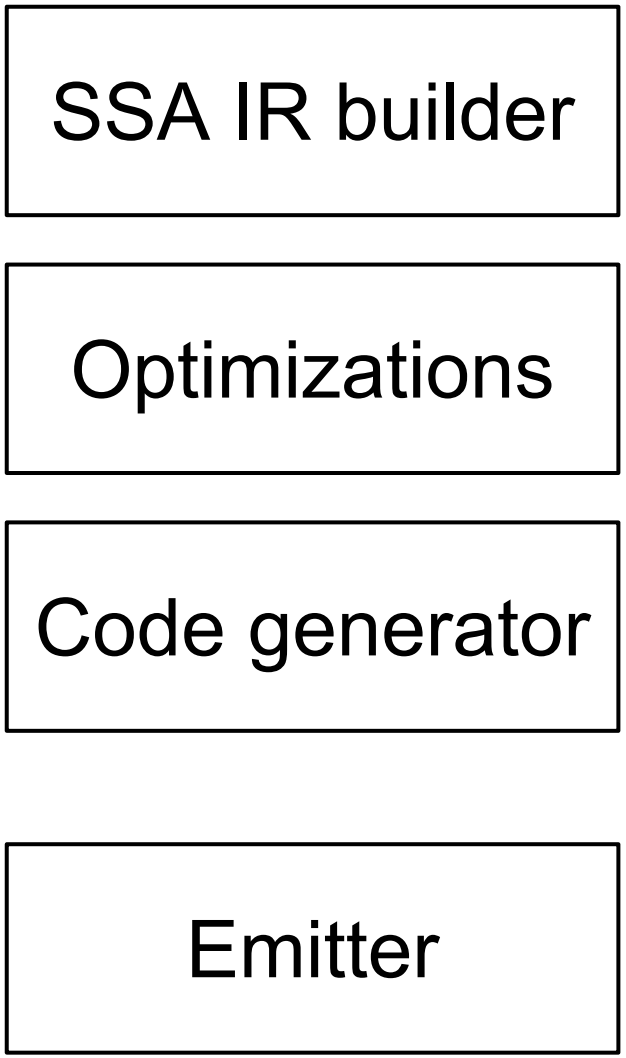
Tree Shaking



Type Inference

- Global fixed point search
- Lattice
  - based on classes
  - (small) unions
  - selected value types
  - special tracking for closures and containers
- Expensive, but produces efficient and small code

Structure of the Compiler: Backend



Creates intermediate representation in SSA from resolved AST.

Performs optimizations on the SSA IR.

Translates from SSA IR to JavaScript.

Emits code for the required functions, structural information and setup code.

Code Generation

Code Generation

- Goals
  - Generated code correctly implements specified semantics
  - Runs as fast as hand-written JavaScript
- Dart has additional dynamic checks
  - Calling with wrong number/names of arguments yields runtime error
  - Lists access is checked against bounds
- Not supported by JavaScript's types!

Object Model

- Dart objects are JavaScript objects
  - Methods and fields are defined as in JS
  - Instances are created using a constructor function
  - Superclass relation defined by prototype chain
- This model is common in applications and well optimized

Object Model

- Object model is setup at load time
- Constructor functions are generated
  - Dart constructors compute field values and call the constructor function
  - Prototypes for setup using the superclass name
- Superclass fields are “copied”
  - Fields are set in the constructor function
  - Stable prototypes and fast objects

Class Encoding

```
class B {
  int i;
  B();
  B.value(this.i);
}

class C extends B {
  C() : super.value(42);
}

main() {
  print(new B());
  print(new B.value(3));
  print(new C());
}
```

```
["", "...", , S, {
  "^": "",
  main: function() {
    P.print(new S.B(null));
    P.print(new S.B(3));
    P.print(new S.C(42));
  },
  B: {
    "^": "Object;i"
  },
  C: {
    "^": "B;i"
  }
},
],
```

Class Encoding

```
class B {
  int i;
  B();
  B.value(this.i);
}

class C extends B {
  C() : super.value(42);
}

main() {
  print(new B());
  print(new B.value(3));
  print(new C());
}
```

```
["", "...", , S, {
  "^": "",
  main: function() {
    P.print(new S.B(null));
    P.print(new S.B(3));
    P.print(new S.C(42));
  },
  B: {
    "^": "Object;i"
```

The trivial constructor for `B`,

```
S.B$B(i) {
  new S.B(i);
}
```

has been inlined (as well as the constructors for `B.value` and `C`)!

```
],
```

Functions

```
int foo(a, b) { … };
```

Functions

```
int foo(a, b) { … };
```

```
function(a, b) { …  }
```

Checking Argument Count

```
int foo(a, b) { … };
```

```
function(a, b) {
  if (arguments.length != 2) throw …;
   …
}
```

Checking Optional Arguments

```
int foo(a, [b]) { … };
```

```
function(a, b) {
  if (arguments.length != 1 &&
      arguments.length != 2) throw …;
   …
}
```

Checking Named Arguments

```
int foo(a, {b}) { … };
```

```
function(a, ?) {
  if (?) throw …;
   …
}
```

Generating Function Calls

- No problem for static and top-level functions
  - Resolution can check call-sites
  - Emit throw instead of call
- Need to encode signatures for methods and closures

Generating Function Calls

- Find all selectors used to call a function named `foo`
- Compute unique names for each selector
- Install a throwing stub for each name on Object
- Redefine valid stubs in subclasses which complete the call

Generating Function Calls

- One possible encoding
  - Append **`$n`** where **`n`** is the number of positional arguments
  - Append **`$a`** for each provided named argument in alphabetical order
- Redefined stubs provide the default value for missing arguments
- JavaScript engines can inline stub calls

Generating Call Stubs

```
class C {
  f(a, {b, c: 1}) { … }
}
…
e.f(1, b: 3);
e.f(1, 2);

On Object:
f$1$b = function(a0, a1) { throw …; }
f$2 = function(a0, a1) { throw …; }

On C:
f$1$b = function(a0, a1) { return f$1$b$c(a0, a1, 1); }
```

Generating Calls to Closures

- Closures are encoded as classes
- Use same encoding for `call` method
- Methods can be extracted from objects as closures
  - In Dart, `e.f(...)` means the same as `(e.f).call(...)`
  - Global inference allows to emit optimal call sites in most cases

Interacting with Native Objects

- Representing integers, lists and strings in Dart is too slow!
- Use JavaScript types
  - optimized by all JS engines
  - add additional checks
  - map Dart to JavaScript method names
- We have to intercept all calls to potentially native targets!

Intercepting Calls on Native Objects

```
foo(x) => x + 1
```

Intercepting Calls on Native Objects

```
foo(x) => x + 1


var foo = function(x) {
  return x + 1;
}
```

Static Interceptors

```
foo(x) => x + 1


var foo = function(x) {
  return add(x, 1);
}
```

Static Interceptors

```
foo(x) => x + 1
```

```
var foo = function(x) {
  return add(x, 1);
}

var add = function(x, y) {
  if (isNumber(x) && isNumber(y)) {
    return x + y;
  } else if (isNumber(x)) {
    throw new ArgumentError(y);
  } else if (!isDartObject(x)) {
    return x.add$1(y);
  }
  throw noSuchMethodException(x, "+", y);
}
```

Static Interceptors

- Allow type safe use of native objects
- Multiple operations on the same operand lead to repeated checks
- Hard to use for optimizations, since implementation is in JavaScript
- (Earlier) Solution: bailout functions
  - This is how virtual machines like V8 and the Dart VM optimize

Bailout Functions

- Assumption: the receiver of +, -, [], ... is most likely of native type
- Propagate this type through the graph and insert type guards
- Compile bailout version
  - supports jumping into appropriate from every type guard
  - called from type guards when check fails
  - uses static interceptors

Bailout Functions

```
foo = function (s, i) {
  var t0, t1, t2;
  t0 = s.length;
  print(t0);
  t1 = t0 - 1;
  print(t1);
  t2 = i + 1;
  print(t2);
  return s.substring(t1, t2);
}
```

`print` statements are used to indicate side-affects.

Bailout Functions

```
foo = function (s, i) {
  var t0, t1, t2;
  if (!isString(s)) return foo$bailout(0, s, i);
  t0 = s.length;
  print(t0);
  if (!isNumber(t0)) return foo$bailout(1, s, i, t0);
  t1 = t0 - 1;
  print(t1);
  if (!isNumber(t1)) return foo$bailout(2, s, i, t0, t1);
  t2 = i + 1;
  print(t2);
  return s.substring(t1, t2);
}
```

Bailout Functions

- Very efficient on benchmarks
- Disadvantages
  - Produce more code
  - Complicated to compute bailout environments
  - Depends on heuristics
- Better solution: create interceptor objects for specific types
  - interceptors can be shared through the IR
  - global analysis reduces amount of checks

Interceptor Classes

```
class JSArray<E> extends Interceptor implements List<E>, JSIndexable {
  /// Returns a fresh growable JavaScript Array of zero length length.
  factory JSArray.emptyGrowable() => new JSArray<E>.markGrowable(JS('', '[]'));

  void add(E value) {
    checkGrowable('add');
    JS('void', r'#.push(#)', this, value);
  }

  E removeAt(int index) {
    if (index is !int) throw new ArgumentError(index);
    if (index < 0 || index >= length) {
      throw new RangeError.value(index);
    }
    checkGrowable('removeAt');
    return JS('var', r'#.splice(#, 1)[0]', this, index);
  }
}
```

Full code at available at code.google.com.

Interceptor Classes

```
class JSArray<E> extends Interceptor implements List<E>, JSIndexable {
  /// Returns a fresh growable JavaScript Array of zero length length.
  factory JSArray.emptyGrowable() => new JSArray<E>.markGrowable(JS('', '[]'));

  void add(E value) {
    checkGrowable('add');
    JS('void', r'#.push(#)', this, value);
  }

  E removeAt(int index) {
    if (index is !int) throw ne
    if (index < 0 || index >= l
      throw new RangeError.value(index);
    }
    checkGrowable('removeAt');
    return JS('var', r'#.splice(#, 1)[0]', this, index);
  }
}
```

The special JS call is recognized by the backend and allows to include arbitrary JavaScript code.

Full code at available at code.google.com.

Interceptor Classes

- Special treatment in the compiler
  - Methods take additional receiver argument
  - this is rewritten to refer to that argument
- Call sites have Dart semantics!

Use of Interceptors

```
getInterceptor = function(receiver) {
  if (typeof receiver == "number") {
    if (Math.floor(receiver) == receiver) return J.JSInt.prototype;
    return J.JSDouble.prototype;
  }
...
  if (receiver.constructor == Array) return J.JSArray.prototype;
  return receiver;
};
getInterceptor$ns = function(receiver) {
  if (typeof receiver == "number") return J.JSNumber.prototype;
  if (typeof receiver == "string") return J.JSString.prototype;
  return receiver;
};
J.$add$ns = function(receiver, a0) {
  if (typeof receiver == "number" && typeof a0 == "number")
    return receiver + a0;
  return J.getInterceptor$ns(receiver).$add(receiver, a0);
};
```

Other Challenges

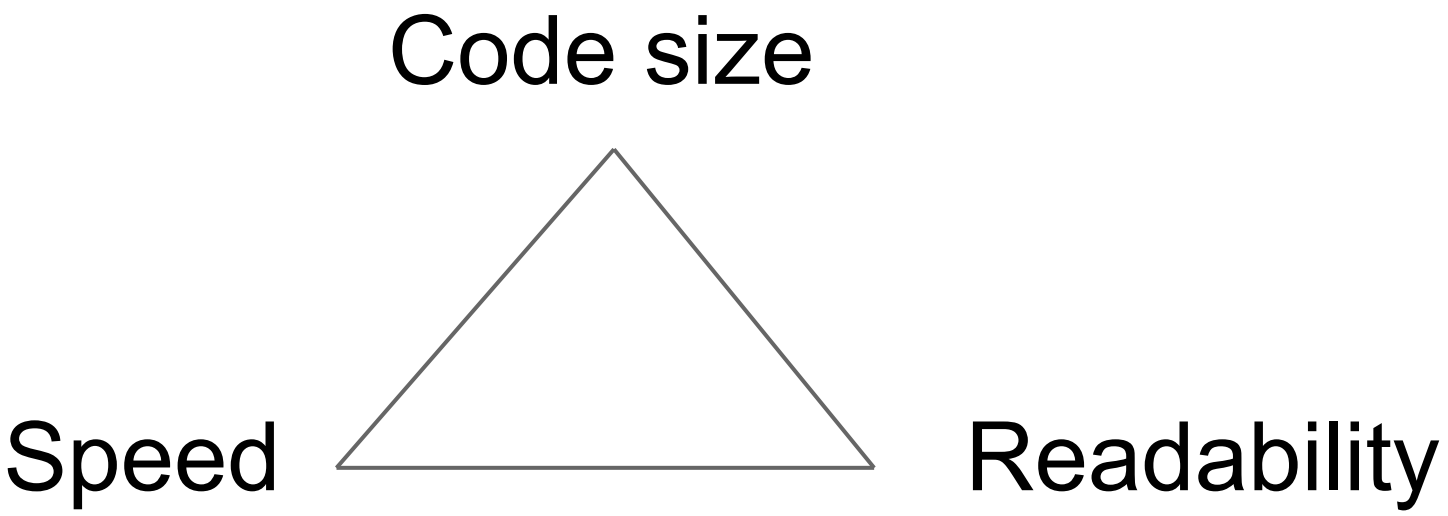
- No integers in JavaScript, while Dart has real integers
  - Emulation is very slow
  - Use doubles and provide compatibility mode in VM to find bugs
- Emitting idiomatic code after optimizations
  - Uncommon patterns use slow paths in JavaScript engines
  - Some IR constructs translate back poorly

Other Challenges

- JavaScript engines change behavior over time
  - Code inside a function used to be slow for some time
  - The empty string property forced objects to be slow

Development Process

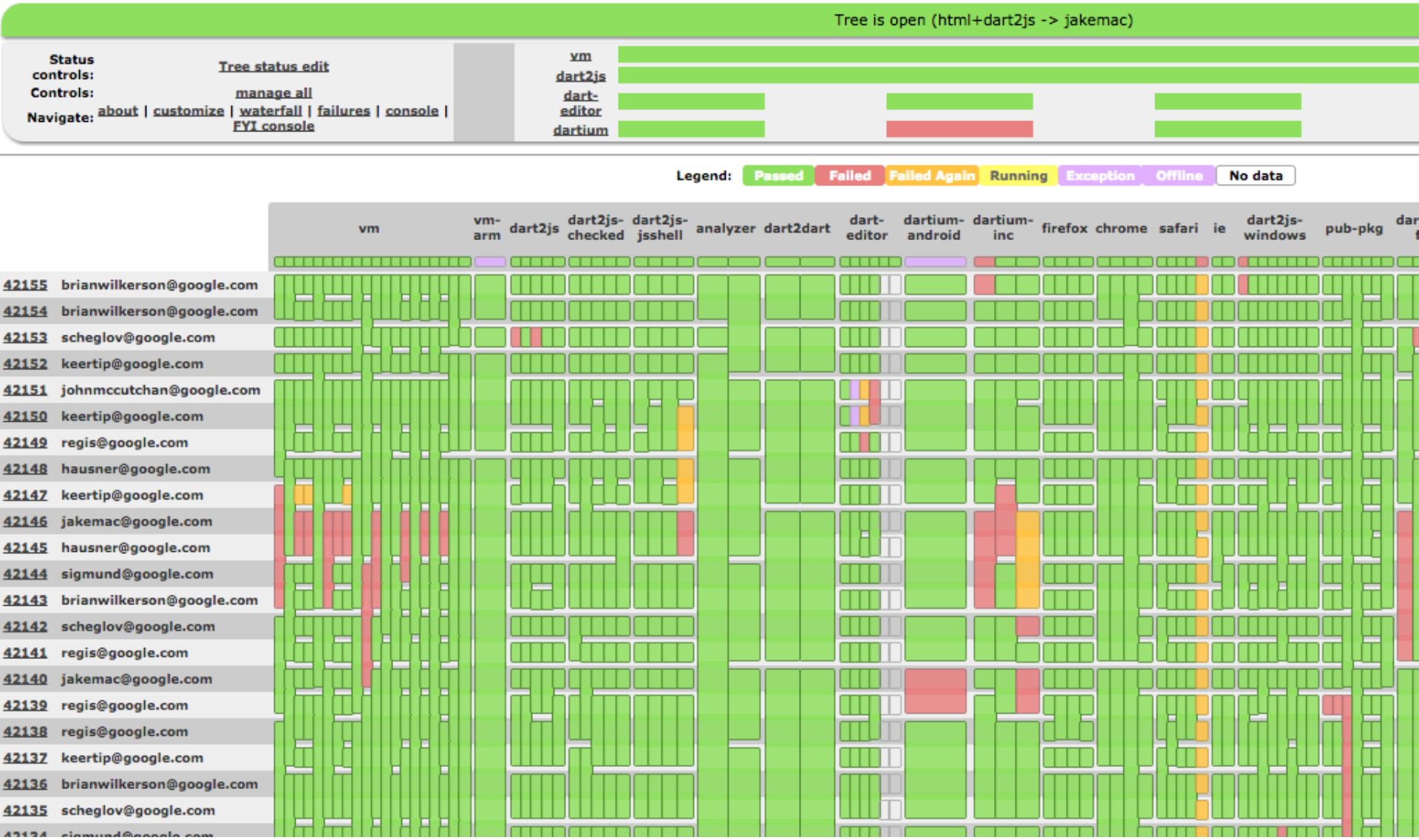
What to Optimize?



Development Process

- Small teams per project but many dependencies
  - Libraries
  - Compilers
  - HTML interaction
  - Testing
  - ...
- Automated testing and code reviews

Automated Testing



Dart Buildbot

Testing in Dart

- Test infrastructure support is a full-time job
- Hardware and time intensive
  - ~14000 tests
  - ~42000 commits
  - Tested on different combination of CPUs, browsers, and modes

Testing Compilers

- Testing a compiler is difficult
- End-to-end tests may pass due to other bug or state of the compiler
  - Testing against artifacts after each phase needs more infrastructure/maintenance
  - Negative and generated tests help
- Language tests should be written strictly from the specification

Current Developments

Asynchronous Programming

- Dart is single threaded
  - DOM events are scheduled in a queue
  - IO events are non-blocking
- Future: promise to complete with value/error
- Streams: subscribe to be informed of events

Asynchronous Programming: Futures

```
void printDailyNewsDigest() {
  File file = new File("dailyNewsDigest.txt");
  Future future = file.readAsString();
  future.then((content) => doSomethingWith(content))
        .catchError((e) => handleError(e))
        .whenDone(() => file.close);
}
```

Asynchronous Programming: Streams

```
var subscription = button.onClick.listen((mouseEvent) {
    clickCount++;
     // unsubscribe after the third click
    if (clickCount == 3) {
      subscription.cancel();
    }
 });
```

Asynchronous Programming

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

Asynchronous Programming

```
readWrite() {
  read((c) {
    write(c, handleError);
  },
  handleError);
}

// Finally block cannot be handled.
// Easy to make mistakes in error
// handling.
// … and fairly unreadable.
```

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

Asynchronous Programming

```
readWrite() {
  Future f = read();
  return f.then((c) => write(c))
          .catchError(handleError)
          .whenComplete(close);
}

// Control flow must be dealt with in
// library.
// Chaining of futures is tedious.
```

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

Asynchronous Programming

```
readWrite() async {
  try {
    var c = await read();
    await write(c);
  } catch (e) {
    handleError(e);
  } finally {
    await close();
  }
}

// await suspends the
// activation in a
// non-blocking way!
```

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

Asynchronous Programming

```
import "dart:html";

main() async {
  var context = querySelector("canvas").context2D;
  var running = true;    // Set false to stop game.

  while (running) {
    var time = await window.animationFrame;
    context.clearRect(0, 0, 500, 500);
    context.fillRect(time % 450, 20, 50, 50);
  }
}
```

Current Developments in dart2js

- New, CPS based, intermediate representation
- async/await support
- Shared frontend with analyzer
- New code emitter

Try Dart

- dart2js in the browser: http://try.dartlang.org/
- Dart is available under a BSD license
- Online resources
  - Website - http://www.dartlang.org/
  - Code - http://dart.googlecode.com/
  - Libraries - http://api.dartlang.org/
  - Specification - http://www.dartlang.org/docs/spec/