Performance & Optimisation on Scaja.js

Damien Engels, Tristan Overney
Sébastien Doeraene, Martin Odersky (Supervisors)

January 20, 2016

Chapter 1

Introduction

Nowadays, a huge part of our computer utilisation is through a web browser and web apps. Developping such apps is most commonly done in JavaScript as it is pretty much universally supported. Scala.js allows you to develop those kind of programs without the hassle of using JavaScript directly, with the type safety of your regular Scala and still keep a great inter operability with the many existing JavaScript libraries. Since its conception, Scala.js has seen a lot of optimization and rewriting done in the sole purpose to increase its performances and reduce the size of its generated code. In this project, we were tasked to first implement or port a benchmarking framework to Scala.js to have a reliable way to compare two implementations of the same feature and be able to tell which one is the best in a relatively effortless manner. After that we took a deep dive in the Scala.js tools and were tasked to implement several optimisation to the generated javascript code. We first implemented the constant folding for the binary operator *String_+*, described in chapter 3. Then we rewrote the instance checking with a tag system and eventually we worked on the inlining constructor in specific cases; both in order to reduce the amount of method calls.

Chapter 2

Scalameter for ScalaJS

Chapter 3

String_+ constant folding

Chapter 4

Instance test optimisation

Chapter 5

Constructor inline on effectively final classes

5.1 Description of the optimisation

Scala allows for multiple constructors per classes which is not the case for JavaScript. Up until now, ScalaJS solved that problem by only having the JavaScript constructor initialize the class fields with the zero of their type and then have a method *init* per constructor which actually does what its corresponding Scala constructor does. This means that currently if you want to create an instance of type A the outputted JavaScript code will first create an instance of type A with the "empty" constructor and then call the init method on it.
But in most cases, a Scala class doesn't have more than one constructor and if that class is effectively final (has no subclasses with instances), otherwise there would be issues in the subclasses with the call to the optimized parent constructor. In those cases, it would be possible to get rid of the init method and directly "inline" the Scala constructor instructions in the JavaScript constructor, right after initializing the field s with zeros of their type. This optimisation would reduce the amount of code generated and reduce the number of method call executed thus improving both the file size and the running time. Example 0 shows the Scala code for an optimizable class and an example of call site; example 1 shows the current way of defining such class; example 2 shows the generated code we want to build when inlining the constructor. (these examples do not show all the generated JavaScript code, only the relevant parts)

Listing 5.1: Example Scala Code.

```
class A(message: String) {
    def tell(): Unit = println(message)
}
```

```
...

val someA = new A(someString)

...
```

Listing 5.2: Generated JavaScript without optimisation.

```
/** @constructor */
function $c_LA() {
  $c_O.call(this);
  this.message$2 = null
}

/* Skipping prototype shenanigans ... */

$c_LA.prototype.init___T = (function(message) {
  this.message$2 = message;
  $c_O.prototype.init___.call(this);
  return this
});

...

var someA = new $c_LA().init___T(someString);

...
```

Listing 5.3: Generated JavaScript with constructor optimisation.

```
/** @constructor */
function $c_LA(message) {
  $c_O.call(this);
  this.message$2 = null;
  this.message$2 = message;
  $c_O.prototype.init___.call(this);
}

/* Skipping prototype shenanigans ... */

/* No init method! */

...

var someA = new $c_LA(someString);

...
```

5.2 Implementation Details

This implies changes in the emitting phase, when the Compiler IR is desugared into JavaScript IR. The first step is to go through all classes and find those who only have one constructor, we count the methods of the class where their name is a constructor name (starting with "init___"), and are effectively final, we look at the parent of each classes that has instances, those parent are not effectively final. This allows us to build a set with all the classes that uses the constructor Optimisation. Then we have to have a different behavior for the following point in code emitting:

- when building the constructor of a JS Class; if it uses the optimisation we have to fetch the "init" method from the methods of the class, change the parameter list and add the statement from the fetched method without its return statement as it is now part of the constructor.
- when generating all the method code we have to filter out the "init" method which is no longer relevant and should not appear in the generated code.
- when desugaring a function, each time we encounter a call site to a new we have to check wether to call the constructor only or the "empty" constructor and then an "init" method.

5.3 Complications & restrictions

While implementing this optimisation we faced different issues which required restrictions or rethinking of our initial or intermediate solution. This section describe the nature of the problem and shows how we restricted or changed the implementation of our optimisation.

5.3.1 Exported or Hard-coded classes

5.3.2 ECMAScript 6 Strongmode

Scala doesn't have any restriction regarding the instruction you can execute in a constructor whereas the specification for ECMAScript 6 Strongmode only allows a limited set of instruction to be done inside a constructor. As such if the selected output mode is ECMAScript 6 Strongmode then we never do the optimisation and keep the "init" method.

5.3.3 Incrementality

Caches! Invalidate caches while doing more caches :)

Chapter 6

Going further