

Performance & Optimisation on Scaja.js

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

January 21, 2016

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 have a flawless interoperability with the many existing JavaScript libraries [1]. 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.

2 Scalameter for ScalaJS

3 *String_+* constant folding

4 Instance test optimisation

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 an "init" method 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 fields with zeros of their type. This optimization would reduce the amount of code generated and reduce the number of method calls executed thus improving both the file size and the running time. The code snippet 1 shows the Scala code for an optimizable class and an example of call site; code example 2 shows the current way of defining such class; finally, code snippet 3 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 1: Example Scala Code.

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

...

val someA = new A(someString)

...
```

Listing 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___T.call(this);
    return this
});

...
```

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

...
```

Listing 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 Hard-coded classes

Some of the Scala.js environment is specified in an hardcoded file named *scala-jsenv.js* (or an equivalent file for EcmaScript 6 Strong Mode) most of the classes instantiated in that file (e.g. *jLNullPointerException*) do not always have all their constructor present in the generated code, therefor they won't always be optimized and it is not possible to hardcode either version of the constructor call site. We chose to exclude those classes in the form of a blacklist when computing the set of classes which will be optimized. Only one class instantiated in that file was always optimized and its call site was changed in the hardcoded file to match the optimized form.

5.3.2 Exported classes

Optimizing exported classes is problematic because it requires the code importing our classe to have knowledge whether the class has been optimized or not. For this reason, we do not optimize exported classes.

5.3.3 EcmaScript 6 Strong Mode

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

5.3.4 Incremental compiler

Caches! Invalidate caches while doing more caches :)

6 Going further

Acknowledgments

We thank Sébastien Doeraene for helping us all along this project, taking us in a deep dive into Scala.js and being patient even

References

- [1] Sébastien Doeraene. Semantics-driven interoperability between scala.js and javascript. Submitted for review to ECOOP '16, 2015.
- [2] Aleksandar Prokopec. Scalameter. <http://scalameter.github.io/>, 2012.
- [3] Michel Schinz. Object-oriented languages. Slides on relative numbering taken from the Master's course Advanced Compiler Construction, 2015.