Performance & Optimisation on Scala.js

Damien Engels, Tristan Overney
Sébastien Doeraene, Martin Odersky (Supervisors)
{*firstname.lastname*}*@epfl.ch*

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 [2]. 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.

2 Scalameter for Scala.js

One of the main purposes of a compiler generating code instead of one having to write it by hand is that the compiler can perform a handful of code transformation and optimisation that would be near to impossible to perform by hand. Those optimisation can potentially increase performance and / or reduce the resulting code size. However, there may be many transformation possible at any given point in the compilation process and finding out which one have a real impact on the code can be cumbersome, especially when the effect is subtle.
A great way to measure whether some code transformation is actually a valid optimisation is to benchmark the code against it's transformed version. This can however be challenging when the target platform is a virtual machine. Taking out of the picture all the variation caused by other program executing on the

same machine, there will still be a great deal of variation in the measures performed due to the virtual machine itself. Garbage collection phases for example, might cause some very large outliers to appear in the data. In addition, most VM's perform advanced optimisation techniques (just in time compilation, inlining, dynamic compilation) that cause the performance to improve over time. There already exist a few benchmarking tools for JavaScript virtual machines, however most of them do not take those variations into account, which renders them useless for measuring running times in the order of milliseconds. For this reason, our first task in this project was to port the ScalaMeter framework (a benchmarking framework for the JVM) so that it could compile with Scala.js.

3 +[string] constant folding

3.1 Description

The first optimisation we were tasked to implement was constant folding for the binary operator *+[string]*. which means replace *+[string]* expression where both sides are literals (which could be any literal) into the value produced by that expression. E.g. if you have the following expression: *"hello " + "world"*, you can transform it, at compile time, into its result: *"hello world"*. This means you have one less *+[string]* operation to do at runtime. It also reduces the code size by a few characters.
There are a few other cases that can be optimised (the double arrow indicates the transformation done by our constant folding):

- *"" + (a: String) =>a*: if we add the empty String to any value typing to String then we can simply put the value itself.
- *(a + "hello ") + "world") =>a + "hello world"*: we can extract the two constant and "flatten" the *+[string]* operation.
- *(a + "") + b) =>a +[string] b*: if the empty string is between two *+[string]* we can remove it if we make sure the + operator of the new expression is a *+[string]*.

3.2 Additional remarks

In most cases the Scala.js compiler will run on the JVM and not bootstrapped in JavaScript, this introduces a slight issue as the JVM and EcmaScript do not have the same specifications regarding the String representation of numbers [1]. Our first approach was to implement the EcmaScript specification in the *OptimizerCore* but the code was huge for those relatively small cases, thus it was decided not to constant fold numeric literals which would require the use of that big specification function.
In the mean time we had written unit tests for pretty much all cases of the EcmaScript specification algorithm and suddently the Scala.js testSuite was always failing when doing the fullOpt but succeeding in fastOpt. It was soon

discovered that the Google Clojure Compiler, also running on the JVM and only used in the fullOpt mode, had a bug because it did constant folding (in the JVM) but did not follow the EcmaScript specification for printing numerics! An issue has been opened on the Clojure Compiler Project and has yet to be resolved despite a pull request of ours. [3]

4 Instance test optimisation

4.1 Motivation

Scala, like most object oriented languages has a construct that allows programmatic membership tests. Idiomatic Scala depends heavily on this feature, especially though the use of pattern matching. The JavaScript virtual machine has support for membership test (the *instanceof* operator), but it's behaviour is semantically different from Scala's *isInstanceOf*[T] method, therefore forcing Scala.js to provide some runtime support.
The current implementation relies on a field $*classData* present at runtime in the prototype of each class. This field contains among other things, information about the class' ancestors as well as weather it is an instance of the ArrayClass and it's depth. The ancestors field is a JavaScript object where the keys are the name of the ancestors and the values are all set to 1, so that an instance test is just a lookup in the object to check whether the key is present.

Listing 1: Old membership test

```
function $is_F0(obj) {
  return (!(!((obj && obj.$classData) &&
            obj.$classData.ancestors.F0)));
}
```

This can be quite slow for several reasons, the main on being that it needs to lookup fields in 2 objects (*ancestors*, and then $F0$).

4.2 Description of the optimisation

To improve the efficiency of the membership test, we assign a unique numerical identifier to each class in the program. We then generate for each class an integer array filled with all 0 except for all the indices of the class' subtypes. The membership test is then reduced to an array lookup, which is much more efficient than an field lookup in an object.

Listing 2: New membership test

```
var $Is_F0 = [1,0,0,0,1,...,0,0,1];
```

```
function $is_F0(obj) {
  return (!(!(obj && $Is_F0[obj.$typeTag])));
}
```

4.3 Further optimisations

The main problem with the previous implementation, is that it increases dramatically the size of the generated JavaScript program. To attenuate this effect, we extend the Scala.js runtime support to include a *expandSubtypeArray* function that will construct the array at runtime from a list of intervals.

Listing 3: Compressed subtype array

```
var $expandSubtypeArray = function(tags){
  var expanded = [];
  var len = 0;
  for (var i = 0; i < tags.length; i++) {
    var interval = tags[i];
    var start = interval[0];
    var end = interval[1];
    while(len < start){ len = expanded.push(0); };
    while(len <= end){ len = expanded.push(1); };
  }
  return expanded;
};

var $Is_F0$ = $expandSubtypeArray([[13, 13], [21, 34]]));
```

We also try to use relative numbering [5] to map classes to their tags, such that in most cases, the subtype array doesn't even need to be generated. In those cases, the membership test is reduced to a mere few comparisons.

Listing 4: Constant folding of subtype array

```
function $is_F0(obj) {
  return (!(!(obj && ((obj.$typeTag === 13) ||
      ((obj.$typeTag >= 21) && (obj.$typeTag <= 34))))))
}
```

4.4 Additional remarks

The scheme presented above works well for static membership tests, when all the classes are known at compile time. It doesn't work for dynamic membership tests, or when new classes are created at runtime. Fortunately, it can be extended to work in these cases too, as presented in this section

4.4.1 Array types

The only types that can be created at runtime in the Scala.js environment are the array types. To handle array, the previous implementation stored the baseType as well as the depth of the array in the $*classsData* field. We extend our previous implementation to accommodate array types by the following encoding of the $*tagData* field: The first bit is reserved to discriminate array types from other types, this makes it easy to check for array types as all of their tags are negatives. The next 8 bits are reserved for the depth of the array. This is sufficient since the maximum depth allowed for multidimensional arrays in JavaScript is 256. The 23 bits left are then used to encode the array's component type tag. This reduces the maximum number of compile-time types that we can encode to roughly 8 millions. However, since a high number of types also means a very large JavaScript program, reaching this limit is very unlikely before the program reaches a problematic size.

4.4.2 Dynamic membership tests

Because the transformation presented above will not work for dynamic membership tests, we cannot discard the $*classData* object and its *ancestors* field entirely. However, since dynamic tests are a lot less frequent, we can migrate all those object to a program-global data structure indexed by each class' tag. Furthermore, since there is a one-to-one mapping from classes to tags, we can compress the ancestors object to an array of corresponding tags. As we see later, this reduces the size of the output program enough to counter balance the increase caused by the subtype arrays.

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, Scala.js solved that problem by only having the JavaScript constructor initialise 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. 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 initialising the fields with zeros of their type; but if and only if that class is effectively final (has no subclasses with instances), otherwise there would be issues in the subclasses with the call to the optimised parent constructor. 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. The code snippet 5 shows the Scala code for an optimizable class and an example of call site; code example 6 shows the current way of defining such class; finally, code snippet 7 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: Example Scala Code.

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

...

val someA = new A(someString)

...
```

Listing 6: 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;
  return this
});

...

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

...
```

Listing 7: Generated JavaScript with constructor optimisation.

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

```
/* 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 behaviour 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 Additional remarks

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. *jl_NullPointerException*) do not always have all their constructor present in the generated code, therefor they won't always be optimised 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 optimised. Only one class instantiated in that file was always optimised and its call site was changed in the hardcoded file to match the optimised form.

5.3.2 Exported classes

Optimising exported classes is problematic because it requires the code importing our class to have knowledge whether the class has been optimised or not. For this reason, we do not optimise 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 optimisation and keep the "init" method.

5.3.4 Incremental compiler

Last but not least, the naive implementation described above doesn't take into account the fact that the Scala.js compiler works in an incremental manner; a class/method will go through the full compiler pipeline only if it has been modified. But with the optimisation described above, we might optimise class A (taken from example 5) in a run and then, during the next compile run our program might have a new class B extending our class A; which means we will no longer optimise the class A. This will break everything as all the statement where we instantiate the class A might be in a class/method where nothing has changed since the last run and as such the callsite to A constructor will not be changed back to the non-optimized version. To correct this issue we used the same idea that was already implemented on the *OptimizerCore*: keep track of the methods each time they ask to know if a class uses the constructor optimisation. This is done through a new class: the *IncClassEmitter*; it creates and control the ScalaJSClassEmitter and keeps the decision on whether or not a class uses the constructor optimisation while keeping a map of each classes have been asked about and a list of who asked. This enable us, at the beginning of each run, to compute the Set of classes for which their constructor optimisation changed (compared to the previous run) and then invalidate all methods who asked for a class that changed. Then we can proceed with the habitual progression of the compiler pipeline and all the code parts that needed an update will be reprocessed.

6 Performance evaluation

As mentioned above there are two factors we want to minimise when doing optimisation, generated code file size and runtime performance, in this section, we summarise our findings for each of those aspects. In the case of the membership test, we also try to find the optimal number of comparisons from which generating the subtype array is beneficial. This variable is represented by the *limit* parameter, where a *limit* of 0 corresponds to the case where we always generate the subtype array.

6.1 Generated code file size

This was the easiest aspect to test. We looked at two different programs to make comparison. The first one is the historical Scala.js example programme: Reversi.scala which is a good example of a small but real world Scala.js app. And the second one is the Scala.js test suite which generates a huge JavaScript file and thus give us an example of how bigger programs would react.

6.1.1 Methodology

For each of the tested project we simply ran the sbt commands *fastOptJS* and *fullOptJS* successively and looked at the file of the created JavaScript files.

6.1.2 Measures

Figure 1 organises all the different file size measured in Bytes.

	Reversi.scala		Scala.js testSuite	
	fastOpt	fullOpt	fastOpt	fullOpt
Current Scala.js master branch	538’088B	123’708B	21’385’239B	4’347’723B
Constructor inlining branch	519’886B	117’508B	20’669’926B	4’229’335B
Difference with master	-3.38%	-5.01%	-3.34%	-2.72%
isInstanceOf branch (limit=0)	501’495B	120’046B	21’078’724B	4’381’011B
Difference with master	-6.80%	-2.96%	-1.43%	+0.77%
isInstanceOf branch (limit=1)	499’502B	119’672B	20’938’110B	4’368’699B
Difference with master	-7.17%	-3.26%	-2.09%	+0.48%
isInstanceOf branch (limit=2)	499’249B	119’560B	20’933’279B	4’367’894B
Difference with master	-7.22%	-3.35%	-2.11%	+0.46%
isInstanceOf branch (limit=3)	499’249B	119’560B	20’932’831B	4’367’461B
Difference with master	-7.22%	-3.35%	-2.12%	+0.45%
isInstanceOf branch (limit=4)	499’318B	119’492B	20’932’940B	4’367’380B
Difference with master	-7.21%	-3.41%	-2.12%	+0.45%
isInstanceOf branch (limit=5)	499’525B	119’415B	20’933’319B	4’367’029B
Difference with master	-7.17%	-3.47%	-2.11%	+0.44%

Figure 1: Generated JavaScript file size in Bytes for fullOpt or fastOpt with the size difference.

6.1.3 Observation

Both optimisation have a positive effect on the file size of the generated code. Where it seems logical for the constructor inlining it may seem strange at first for the isInstanceOf optimisation but as explained previously, the increase in size caused by the subtype arrays is counter balanced by the fact that we were able to change all of the ancestor fields from objects with long string keys to packed arrays of ints.

6.2 Runtime performance

Gathering valid and relevant data for this criteria was a bit harder than for the file size. As described in section 2, it can be challenging to collect meaningful measures on runtime performace impact of code transformations.

6.2.1 Methodology

We used the ScalaMeter framework, that we previously ported to Scala.js, to measure the impact that our transformations had on the runtime performance. For the final classes optimisation, we created 5 * 1’000’000 object on every iteration of the test, whereas for the instanceOf optimisation, we perform roughly 8 * 1’000’000 membership tests for every iteration. We report our results in the tables 2 and 3 below.

6.2.2 Measures

	Constructor Inlining fastOpt	Constructor Inlining fullOpt
Current Scala.js master branch	113 ms	111.8 ms
Constructor inlining branch	118 ms	36.6 ms
Difference with master	+4.42%	-67.26%

Figure 2: Constructor inlining benchmark reported runtimes in millisecond for fullOpt or fastOpt with the performance difference.

	isInstanceOf fastOpt	isInstanceOf fullOpt
Current Scala.js master branch	229.7 ms	112 ms
isInstanceOf branch (limit=0)	144.3 ms	N/A
Difference with master	-37.16%	N/A
isInstanceOf branch (limit=1)	144 ms	N/A
Difference with master	-37.30%	N/A
isInstanceOf branch (limit=2)	142 ms	N/A
Difference with master	-38.17%	N/A
isInstanceOf branch (limit=3)	139 ms	N/A
Difference with master	-39.48%	N/A
isInstanceOf branch (limit=4)	138.3 ms	N/A
Difference with master	-39.77%	N/A
isInstanceOf branch (limit=5)	141 ms	N/A
Difference with master	-38.61%	N/A

Figure 3: isInstanceOf benchmark reported runtimes in millisecond for fullOpt or fastOpt with the performance difference.

6.2.3 Observation

The first thing that we notice is that there is not a great runtime performance improvement from the constructor inlining optimisation in fastOpt. However, it seems to have a tremendous improvement in fullOpt mode. This might be due to the fact that the code outputted by Scala.js becomes simpler / closer to native JavaScript code, and thus easier for the Google Closure Compiler to optimize.
We could unfortunately not test the runtime performance of the isInstanceOf optimisation in fullOpt due to a bug (most probably in the Google Closure Compiler). We can however already guess that a good value for *limit* would be around 3 or 4. That is 3 or 4 comparisons before deciding to use an array of subtype relations.

7 Conclusion

The first goal of the optimisations we implemented for this project was to improve the performance of the generated code. We managed to accomplish that goal but we were quite impressed by the reduction in generated file size. The only results that we would like to see are the improvements in speed and code size when combining both the isInstanceOf and constructor inlining optimisation! But this should happen soon when both optimisation will be merged together.

7.1 Going further

Because Scala.js version 0.6.6 is really close and implementation tweaks are still needed, the isInstanceOf and constructor inlining optimisation are yet to be merged into Scala.js. The logical next step is to do the required tweaks to both optimisation so they can be part of the next Scala.js release.
Regarding the constructor inlining, we could optimise classes effectively final with more than just one constructor. One way to do so would be to have to inline the primary constructor, just as we currently do when there is only one constructor, and have the auxiliary constructors as static methods calling the primary constructor and then executing its instructions. But more thoughts are needed to find out if it's the best way to proceed.
Regarding the isInstanceOf optimisation, the first thing to do is to implement incrementality as it suffers from the same issues as the constructor inlining optimisation. There is also room for another big optimisation. Code generated from Scala tends to obtain most it's calls to $isInstanceOf$ from pattern matching. By detecting chains of if statement over $isInstanceOf$ calls in the optimiser, we could potentially rewrite them into native JavaScript *switch* statement over the object's tag, eliminating a large amount of method calls.

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 when we had off-by-one semicolons error. We also thank Tobias Schlatter and Nicolas Stucki for their time spent reviewing our pull requests.

Ressources

Virtually all the code that has been written for this project is currently available in the GitHub organisation `http://github.com/SebsLittleHelpers`.

References

[1] Ecmascript 2015 language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 2015.

[2] Sébastien Doeraene. Semantics-driven interoperability between scala.js and javascript. Submitted for review to ECOOP '16, 2015.

[3] Sébastien Doeraene. Wrong constant-folding of the number to string conversion. `https://github.com/google/closure-compiler/issues/1262`, 2015.

[4] Aleksandar Prokopec. Scalameter. `http://scalameter.github.io/`, 2012.

[5] Michel Schinz. Object-oriented languages. Slides on relative numbering taken from the Master's course Advanced Compiler Construction, 2015.