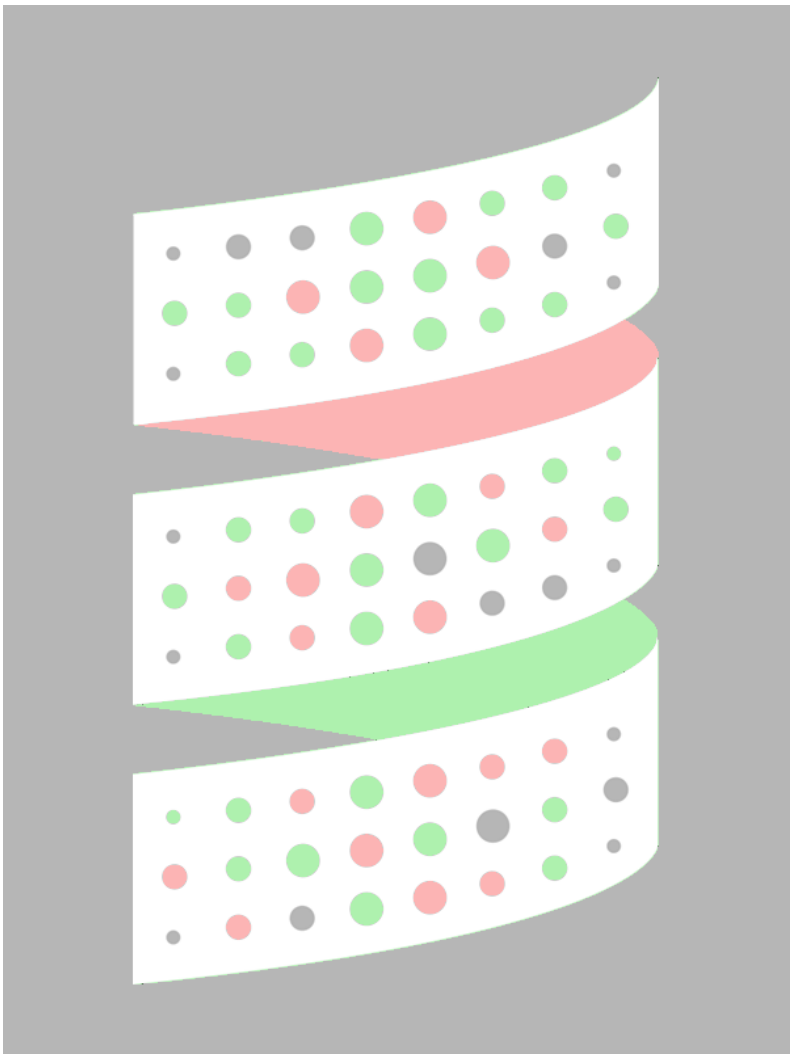
Code Coverage for Dotty



Quentin Jaquier

N° Sciper: 235825

June 2018

Table of Contents

Introduction

Code coverage is an important tool used to measure the quality of a test suite, it gives to the tester some information on which part of the code is covered by the tests, with some statistics. It enables the programmer to define priorities and to help him to improve the quality of his code. There is a lot of coverage tools available for many languages, but currently none of them is implemented for Dotty. Dotty can be seen as an experimental project that aims to simplify the internal of the compiler, having some tools to help the programmer in this task is definitely something that could be useful. The goal of this project is exactly to add code coverage support for Dotty. This work takes some code and inspiration from others code coverage tools, mainly from Scoverage [1], a tool for Scala 2 often used by the community, but implements a new version with some different ideas. There is a lot of different coverage tools available that seems to define their own vision of code coverage, knowing what to expect from such a tool is an important step in the design. This work will present one implementation of a code coverage tool, discussing the different choices, describing the implementations and performances, and proposing some optimization.

Code Coverage

In a general manner, obtaining coverage information for a program can be described as a general procedure [2]. First, we need to identify the target entities in the program. These entities can be of several types, depending on the purpose. Then, we will modify the program to add some **probe** in the code that will capture the execution of the entities. This process is also referred as **instrumentation** in this report. We can now execute this new version of the program to obtain a **trace**, and finally process this trace to obtain the coverage information.

Coverage metrics

As we have seen before, the target entities can be of various types, the choice of what we decide to instrument defines mainly the type of coverage that we will obtain, with different granularity and overhead. When to instrument the code is also an important choice that has some impact on the result. Instrumenting at the byte-code level provide low overhead but loses some information about the source code compared to an earlier instrumentation. These multiple possibilities come with different purposes and overheads. A coarse grain version will give a good overview of the coverage with low overhead while a finer grain will give details information with much more overhead.

Line coverage

Line coverage will simply output for each line if it is executed or not. While simple, this version already provides a good overview of the coverage for the code, and can be efficiently implemented at the byte-code level. The disadvantage is obviously that it will give imprecise feedback if multiple actions are written at the same line. This is not a problem on languages that usually define one action per line, but is not well suited for a functional language like Scala, where it is not rare to see one-line code.

Statement coverage

Statement coverage is a finer grain that we can obtain for code coverage, it instruments single statement at a time. The name of statement is a bit misleading, it does not follow a general definition that we could find in a grammar definition, but every statement coverage tool seems to implement its own version. This complexity comes from the fact that it makes no sense to instrument every single entity that we generally consider as statement. For example, it is useless to instrument a block if we

already instrument all his content, this will lead to multiple instrumentation of the same part of the code adding overhead and skewing the statistics. This work implements a variation of statement coverage that will be described in detail later.

Other types of coverage

Other possible ways of implementing coverage is to consider instructions, branches, methods, objects ... All these versions come with different purposes and overheads that a programmer may want.

```
if(a) {
  if(b) 0 else ((1 + 2 * 3) % 4)
} else {
  5
}
```

Figure 1: example of code that shows the difference between coverage metrics

We can see the difference between metrics in the example in Figure 1. If our test includes the value true and false for *a* and only true for *b*, the results will vary in function of the metric that we choose. The different results can be found in Figure 2. For statement coverage, we consider each literal as individual statement and the condition of the *if* as one statement, hence 50%. If this example only uses literal for simplicity, every statement can be arbitrary, displaying more than 50% coverage can be misleading for the user if he is not aware of the details of his tool.

Line Coverage	Branch Coverage	Statement Coverage
100%	**75%**	**50%**

Figure 2: result of different coverage metrics for code sample in Figure 1

Implementation

If the general idea is simple and can be easily understood, the implementation is not straightforward and needs some effort in order to be implemented correctly. The complexity comes from the fact that there is no clear definition of statement coverage, and that it is not always possible to insert probe where we want, as the environment does not always allow it.

The next parts will describe in more details the procedure implemented for this work, the general idea follows the one presented below. This first implementation may seem naive in some part, we will discuss some optimization later in this report.

Probe insertion

This is probably the most complex step in the process as it will directly define the final result of the coverage. First, we have to choose our coverage metric: my implementation implements a variation of the statement coverage. The majority of the tools for code coverage give a good feedback to know what percentage of the code was executed, but hardly ever a good way to identify why some part of the code was not covered. The goal of this implementation is to be useful for the user in both ways. Short-circuiting of Boolean is a good example to explain this idea that is not always supported in other tools.

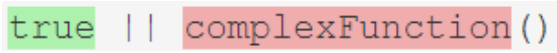


Figure 3: output of the current implementation for Boolean short-circuit

Figure 3 shows the current output of the coverage, in this case, it is clear that *complexFunction()* should not be shown as covered. The union of the two statements into one single covered statement is the kind of misleading output that we want to avoid in our implementation.

New phase in Dotty

I choose to insert probe during the compilation, mapping back to the source code bytes-code instrumentation would be too difficult and potentially lose some information that would be useful for reaching this goal. Before starting to write the new compilation phase, we have to ask us two questions that will influence our work.

Emplacement

In a general manner, it would make sense to implement this work as a plug-in, since we only want coverage information in specific cases. At the beginning of the work, it was not possible to add plug-in to Dotty, I therefore implemented it directly inside the compiler as a new phase that is executed only when some compilation options are added.

The first question that arises when we implement a new compiler phase is to choose where to add it. In the context of code coverage, the earlier is the better in order to not lose any information that we need to give a useful feedback to the user. I choose to add it at the end of the front-end phase, just before the phases dealing with TASTY tree pickling and unpickling. It is important to do it before phases that perform some optimization like dead code elimination. For example, the transformation *FirstTransform* is already removing some unreachable code from *If* tree with constant condition, this is typically some code that we want to instrument before being removed by the different optimization of the compiler as we want to show it as uncovered even if it is removed.

Macro or mini phase

One important difference between a macro and a mini phase is the order of traversal of the tree. The former is a top-down approach while the latter is a bottom-up. Implementing our coverage would be possible with both approaches, but a top-down version is more suitable in order to easily “stop” the instrumentation, when we do not want a branch to be instrumented or when we want to instrument a bigger part of the tree without having the sub-tree instrumented twice.

Type of tree instrumentation

To clearly understand what the tool instruments or not, we can look at how we process each tree of the abstract syntax tree. We can separate the different trees into categories, each member being instrumented more or less the same way.

Fully instrumented

The trees **Ident**, **Literal, New, Super, This**, are fully instrumented as they are. An exception is the Ident wildcard that should not be instrumented as the compiler will not be able to assign a type to it. This is an important choice as it means that there will be potentially a lot of probes inserted, instrumenting them is clearly an arbitrary choice of this implementation. Despite the obvious cost of this choice, the advantage is that we will have a better result of what the real statement coverage is. For example, if we

have an *if* statement without an *else* part, that have a huge computation with 99 literals, a branch coverage metrics would output 50% coverage as the real statement percentage is closer to 99% if we test only this branch. In addition, this choice will enable an easy identification of where a problem has occurred.

Not instrumented

As opposed to the previous category, some of the trees and their branches should not be instrumented, either because the syntax does not support the addition of probes (**Import**), the code will be in any case removed during the compilation (**Annotated**) or because we know that there is nothing interesting to instrument (**Typedtree**).

Recursively transformed

This is the default case, when we have a tree where only his content needs to be instrumented, we can simply recursively continue the transformation. As default case, there is a lot of trees that enter in this category, good examples are **Block** and **Match** as their content will necessarily be instrumented.

Partially instrumented

These are trees that we cannot instrument completely and cannot count only on the instrumentation of their content. For example, **CaseDef** is special in the sense that we do not want to instrument the pattern, but only the guard and the body, and we do not instrument the ID of **PackageDef**.

Select and Apply instrumentation

Select and **Apply** could enter in the category of partially instrumented, but the way we handle them is particularly tricky and cannot be simply described as a binary choice.

```
Object A {                          Object B {
  f(x: Int, y: Int, z: Int)           g(x: Int) = throw new Exception
}                                   }

        A.f(3, B.g(10), complexFunction())
```

Figure 4: sample code that have to be carefully instrumented

Figure 4 shows a pseudo code that illustrates the situation. In this case, we have an object *A* that have one function *f* that takes three arguments, and one object *B* that have only one function that simply throws an exception. During the execution of the last line, the program will first evaluate *A*, that can have some side effect, then evaluate the arguments of the function to finally call the function *f*. Figure 5 shows a coverage output that correctly reflects this execution: it shows *A*, the two first arguments of the function *f* as covered, but not the function *f* itself and the call to function *complexFunction()*. It is interesting to have this kind of feedback since every single statement can generate their own side effects, identifying where the problem comes from can be tricky if it gives incomplete feedback.

```
A.f(3 ,B.g(10), complexFunction() )
```

Figure 5: correct coverage output

In order to reflect this behaviour, it is not possible to instrument the whole apply, as the function *f* would be included in this tree and not shown as uncovered. One solution is to use argument lifting, that

is used during Eta-expansion for example. We can however not use the exact same code; the arguments should be lifted in a precise way in order to reflect the real behaviour.

```
val x0 = A
val x1 = 3; val x2 = B.g(10); val x3 = complexFunction()

x0.f(x1, x2, x3)
```

Figure 6: example of lifting

The idea is to obtain the code of Figure 6 for the line of Figure 5, and to proceed to the instrumentation of this new code, that will also restart recursively the same procedure for the second Apply (*B.g(10)).* The last line can now be instrumented as it is and assigned the position of *f* to be able to display if the function is really executed or not. The current order now reflects the expected execution order, all parts that are executed before the origin or the exception will be shown as covered and not the one after, including *f*. If the argument cannot produce side effects, we do not need to go through this procedure, but we cannot lift only a part of the arguments (for example only the ones that can produce side effect), since it will change the order of the real execution. In our example, if we decide to not lift the first argument, it will not be shown as covered despite the fact that it is executed when the coverage modification is not present.

When we have implemented this trick, we now have to make sure that the original order of the operation is respected in all situations. This is currently not the case: in the case of Boolean short-circuit, Figure 7 shows the current wrong transformation that can happen. In this case, the function *f* will show as covered although it is not in a normal execution.

```
val x0 = f()
true || x0
```

Figure 7: case where the lifting should not be applied

The solution to this problem is to not apply the lifting when we have Boolean operation.

Some additional details should be taken care of related to Select and Apply. The first example is a constraint from the environment: **New** tree should be wrapped in a **Select**, in this situation, we cannot apply our transformation and have to instrument the whole select as one. A second example is the fact that Java classes cannot be used as value, we should therefore stop the recursive transformation and instrument directly the node that uses this kind of module.

```
def T(x:Int)(y:Int)

T(f()) (1)

//Wrong lifting
{
  val $x = f()
  {T($x)}(1)
}

//Correct lifting
{
  val $x = f()
  val $y = 1
  T($x)($y)
}
```

Figure 8: two different lifting that does not reflect the same execution

Figure 8 shows another example of pseudo code that is not correctly supported by the previously described transformation. The lifting at the left is the current result of the process, while the one on the right should be the correct one. The problem comes from the structure of the created tree that creates nested Apply, a naive transformation will transform the code in a wrong way. The solution for this case is to detect this situation and to lift all the arguments to produce the second version of the code.

These details are examples that show the difficulty of this work, one single solution is not always possible, it often works for a general case but is not correct for all possible situations. They are often not complex problems that need a lot of researches to understand and solve, but they are hard to find as they happen in specific situations.

Probe implementation

Once we have identified where we have to add the probes, we have to decide how we want to implement them. The idea is to assign an ID to all trees that we want to instrument. We can then surround every tree with a block containing, before the statement to instrument, a call to an **Invoker** function that we will add in the runtime library of Dotty. The naive version of this invoker will simply write the ID to a file that we will call **trace file**. Optimization that can be done to this step will be discussed later. In Figure 9, we can see an example of instrumentation of the statement print. The invoked function of the Invoker will simply write the ID 4 in the trace file present in the path given as second argument. In addition to this, the process will store some meta information about the tree that is instrumented (position, line number, enclosing method, type of tree …) in order to be used later to generate the report. We will call this file the **statement data file**.

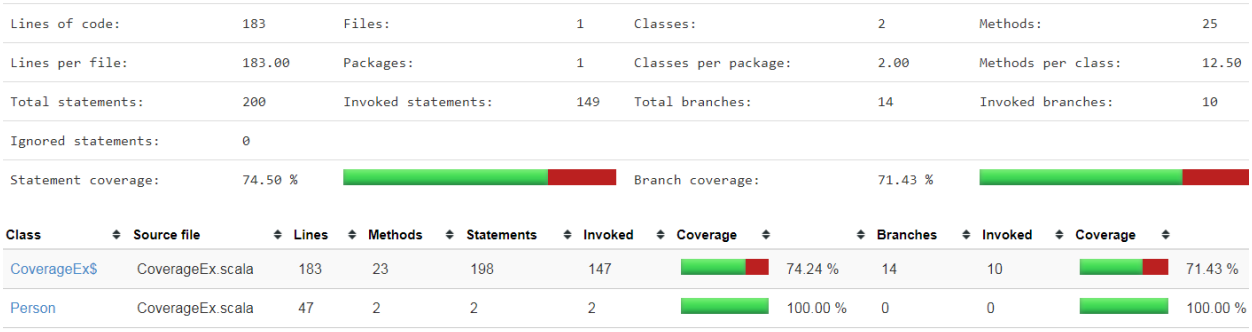
```
{
  dotty.runtime.Invoker.invoked(4, "home\\CoverageData")
  print("Hello world")
}
```

Figure 9: example of instrumentation

Report generation

Once we have a list of statements with the ID's of which one have been executed, it remains to display it to the user in a friendly way. In this step, we could do some interesting analysis, visualization or statistic with these data, choose different output formats, different output platforms ... The advantage at this point is that we have structured data that can be adapted to support many other tools for code quality

inspection. For the sake of time, I simply used the report generator created by Scoverage [1], with some minor adaptation in order to be compatible with my code. It originally creates a HTML report for statement and branch coverage for Scala, the code can therefore be easily adapted to support Dotty. Figure 10 shows an example of report that the implementation currently output.



Lines of code:	183	Files:	1	Classes:	2	Methods:	25
Lines per file:	183.00	Packages:	1	Classes per package:	2.00	Methods per class:	12.50
Total statements:	200	Invoked statements:	149	Total branches:	14	Invoked branches:	10
Ignored statements:	0						
Statement coverage:	74.50 %			Branch coverage:	71.43 %		

Class	Source file	Lines	Methods	Statements	Invoked	Coverage		Branches	Invoked	Coverage	
CoverageEx$	CoverageEx.scala	183	23	198	147		74.24 %	14	10		71.43 %
Person	CoverageEx.scala	47	2	2	2		100.00 %	0	0		100.00 %

Figure 10: example of report

Branch coverage

We have seen previously that sometimes coverage between two metrics is very different and can lead to misleading information, having multiple metrics available can solve this problem. Branch coverage is an important feature for code quality, it can detect a large part of the code that is not executed. Despite the fact that branch coverage is in itself not sufficient to reach our goal, adding it in our implementation comes nearly for free. Currently, we support the instrumentation of any kind of trees, we can therefore instrument additional trees and mark them as branches in the statement data file. We can therefore add a new category to the one described during the Type of tree instrumentation section.

The first tree that we instrument for this purpose is obviously the **If** tree. The current version of the code instruments the *then* and *else* part directly as branches and continues the instrumentation recursively. This may result in some statements that are instrumented twice, once for the statement coverage and once for the branch coverage, but this is not a problem, as long as we make sure to not mix them during the statistics for the report. There is no simple solution that would result in a single instrumentation of such statement. We would want to try for example to instrument the first statement that follows an *If* and mark it as branch, but this is in fact not correct for all cases. For example, in the case of nested *If* statement it is not clear what is the first statement and at which *If* the statement marked as branch refers to. Supporting this kind of problem would require extra work compared to the version that adds a probe to the code.

The second kind of tree that we consider as branch is **Try**. We instrument the expression and the finalizer of the tree if present. This simple addition is a powerful tool that we can add to the final report.

Quality of the coverage

The quality of the result of the coverage is hard to quantify, we cannot really compare two tools since it depends on the purpose of the user. One way to estimate the quality is to show different features that the coverage supports that are not always supported in other tools.

Boolean short-circuits

This is not necessarily the kind of feature that is mandatory for a coverage tool to support, but this is definitely something that can be very useful in identifying some potential bug.

```
true || false
true && false && (true && false)
true && (true && false)
```

Figure 11: example of Boolean short-circuits supported by the implementation

Function application

As discussed in Select and Apply instrumentation, the correct covering of this kind of problem is rare.

```
A.f(3, B.g(10), complexFunction() )   A.f(3, B.g(10), complexFunction() )
```

Figure 12: difference between Scoverage (left) and my implementation (right)

Figure 12 shows the difference between Scoverage and my implementation. Scoverage is also detecting that the call to *complexFunction()* is not made due to the fact that the call to *B.g()* is throwing an exception, but is displaying the function *f* as covered. This may seem like a small difference, but the reality is that the argument is evaluated before the function call, the function *f* is not executed and should not be shown as covered.

Optimization

When using code coverage tools, we cannot avoid some extra execution time due to instrumentation. When we made the choice of statement coverage, we knew that we would add far more probe into the code than any other type coverage, we can therefore expect the extra execution time for our implementation to be larger than the values generally advertised by other type of coverage.

The first place that is impacted by the instrumentation is the compilation time. This process adds an extra phase to the compilation that will add some extra overhead. In order to reduce the overhead, the current implementation uses the power of a top-down approach in order to stop the traversal of the tree when we know that there is nothing to instrument in the subtree, for the different kinds of *TypeTree* for example. If the compilation time is affected by this process, the impact is negligible compared to the overhead that the coverage currently adds to the execution time.

To test the impact of the coverage on the execution time, we will use Scala Community Bench [3], that uses Java Microbenchmark Harness (JMH) to perform some benchmarks. JMH is setup with 2 forks, 5 warm-up iterations and 10 iterations measured. The following values should be taken with care, benchmarking is a hard task that has a lot of external parameters that can skew the results. In addition, the following benchmarks are often thigh loop that are executed many times, this is typically the type of code that suffers from coverage probe. The following results are still useful, mainly to compare the difference between different implementations.

Benchmark	No coverage [ops/s]	Naive coverage [ops/s]	Difference
BounceBenchmark	31900.357	17.83	1789
GCBenchBenchmark	10.259	0.006	1709
TracerBenchmark	1274.965	0.552	2309

Figure 13: benchmark results for the naive version

Figure 13 shows the result of three benchmarks, for each one the code is executed without coverage, and with the naive version of the coverage tool. We can clearly see that, as it is, the tool is unusable due to his huge impact.

Hopefully, there is multiple ways to optimize this process in order to improve the performances. The next part will discuss some possible optimizations that can be applied to our implementation. Due to the time constraints, only a small part is implemented, but it is already sufficient in order to make the tools useable for testing purposes.

Disposable instrumentation

The first idea to reduce the impact comes from a technique called disposable instrumentation [2]. The idea is that, despite that we support duplicates ID in our trace file, we are not interested in the exact number of execution of a statement, we only need to know if a statement is executed once or more, or not at all, for being able to give a coverage feedback to the user. An easy way to translate it to our implementation is the addition of a set in the Invoker that will make sure that one thread will write an ID only once to his trace file. This will not prevent all write of the same ID due to potential concurrent execution, but this will already greatly reduce the number of I/O that is performed.

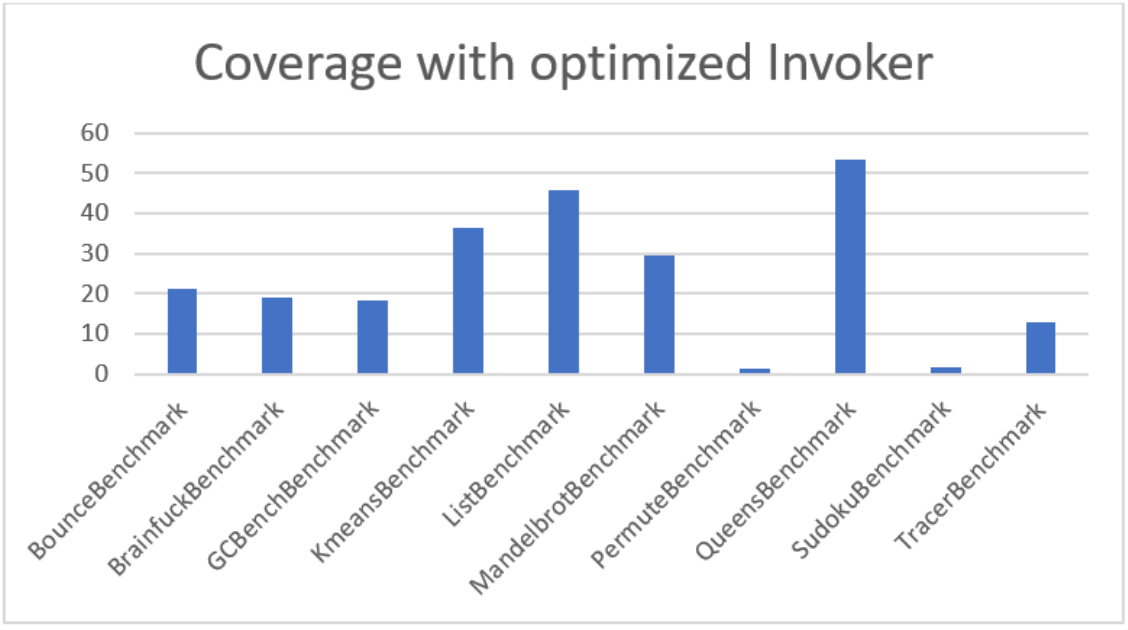


Figure 14: benchmark results for the optimized Invoker

Figure 14 shows the impact on the performance for the version with an optimized Invoker. The impact is now more acceptable, but still has big influence for some benchmarks. We can see that there are two kinds of results: one that has some acceptable overhead (less than 2 times), like *SudokuBenchmark*, and others that have some more noticeable impact. When we look closer to the benchmark that performs poorly, we can see that they all implement a thigh loop that contains a lot of instrumented statements. Even though we do not perform an I/O for every iteration, the call to the Invoker function seems to still have a huge impact.

Performance comparison with Scoverage

The original code for the Invoker comes from Scoverage, but it has been simplified to better fit our purpose. The main difference is that the Scoverage version of the Invoker supports the same thread to write to different folder. Since our implementation only writes the trace file to a single folder, we can save some computation by only checking if a given ID is already present, and not if it is present in a given folder.

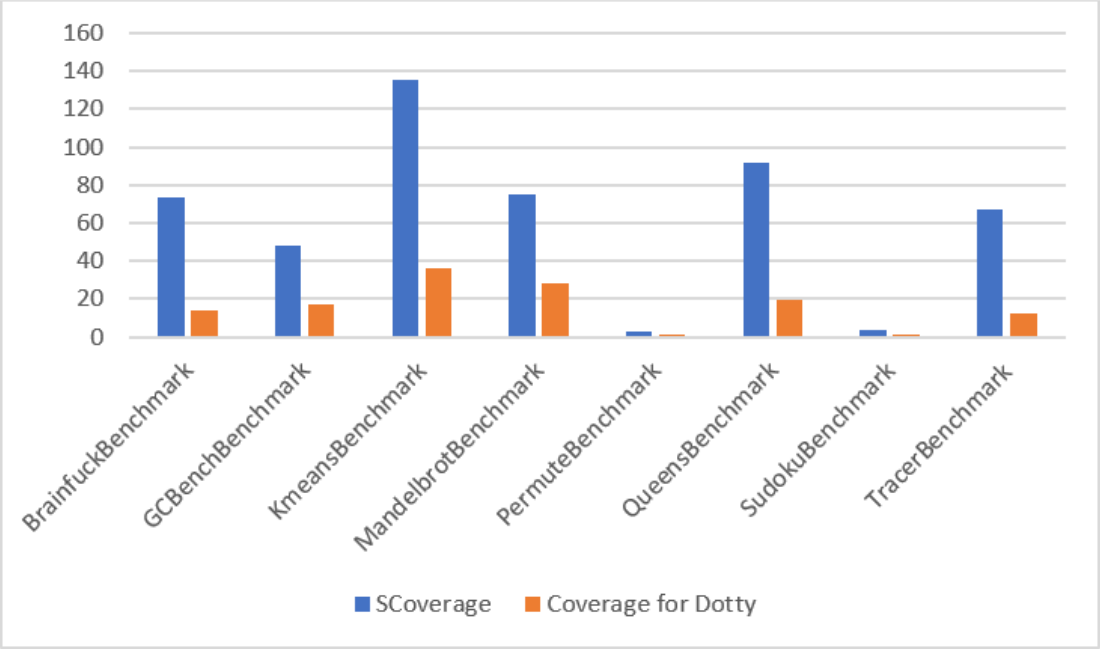


Figure 15: performance comparison between Scoverage on Scala 2 and our implementation on Dotty

Figure 15 shows the current difference of impact compared to an execution without coverage between the current version of Scoverage (1.5.1) with Scala 2 and our implementation with Dotty. Even though we seem to instrument more statement than Scoverage, the performance is still better due to our simplification of the Invoker. We can see that even the tools that seems to be well known to the community also suffer when confronted to this kind of benchmarks. When we look at the Github of Scoverage, they seem to be aware than the Invoker is the bottleneck in some situations, and some pull request have tried to solve this problem, but the project does not seem to have moved since a few years.

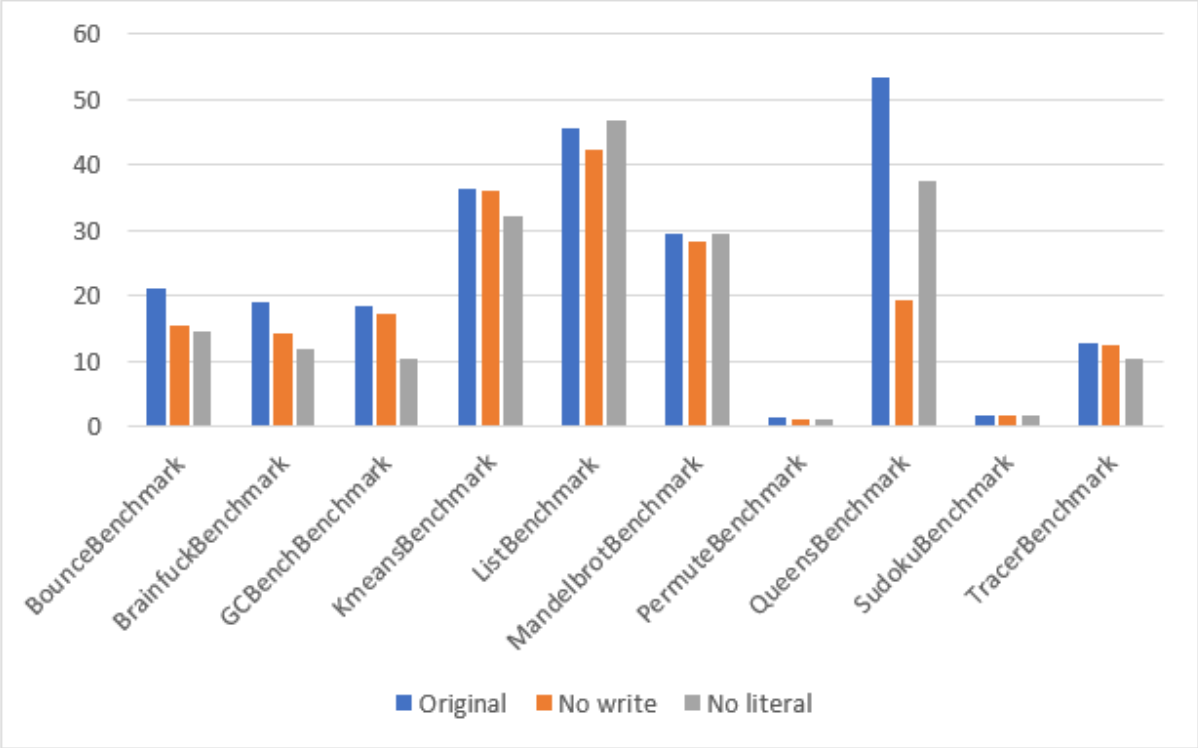


Figure 16: comparison between three versions of the coverage tool

In Figure 16, we can see the impact of the performance if we remove the write to the file from the invoker, the remaining overhead is therefore only the call to the Invoker and the preparation of the write that include the set management. The I/O obviously has a non-negligible impact, but we can still conclude that the bottleneck is more in the set operations of the Invoker than during the I/O. The Invoker part seems to be an unavoidable point to optimize the overhead.

One way to reduce the overhead is to rethink our whole definition of what we have defined in the Probe insertion section. In Figure 16, we can see that if we do not instrument literals, we can reduce some of the overhead. Removing the literals can be justified since they are constant that will never be the source of problem, but we lose the power that we discussed previously. Instrumenting less is always possible, but it will decrease the quality of the output. Another way to optimize the invoker would be to perform this computation in a non-blocking way, by assigning one thread that would be responsible to perform the set computation and the I/O to every thread that calls the invoker.

Disposable instrumentation is a particularly good optimization if we consider the 80/20 rules to be true (80% of the execution times of a program comes from 20% of the code). There is no need to continuously go through the probe's overhead if only one traversal of the 20% of the code is enough. This rule definitely applies for our benchmarks since they all perform heavy computation with only small code. The final goal of disposable instrumentation is to completely remove the call to the invoker. It is particularly interesting in our case since we have seen that the overhead comes principally from these calls, even when we do not perform I/O. We can remove the probe at the application level or at the JVM

level [2]. The former uses some technique that dynamically updates the application, by using a technique called hot-swapping [4]. This technique statically updates the code to enable a dynamic swapping. Our implementation could use this technique to statically prepare the program during the compilation, and support a dynamic swapping in order to remove the call to the Invoker. This is a smart solution, but this technique is created to not need any support from the runtime. Since we have access to it, we could remove the calls at the JVM level, by modifying at runtime the byte-code. This technique could offer very good results, but requires very good knowledge of the implementation of the JVM.

Others optimization

A lot of work has been done in order to optimize the coverage of code. This part will briefly discuss some potential optimization that can be done, it has not the purpose of describing in details how to implement the different optimizations, but to give some ideas for where to start to continue this work.

An important part is to define precisely where probes are needed, to reduce the number of probes, and therefore the number of calls to the invoker. An idea is to use some dataflow analysis in order to identify domination relationship [5]. It can be applied in our code by regrouping in one main statement instrumentation, multiple statements that are always executed if the main statement is also executed. This is not a straightforward task since it would require some important graph analysis to identify these kinds of nodes.

Another article cites that by using software tomography, we only need to instrument 2 % of the branches and still have accurate information [6]. This technique can probably not be used as it is, and the percentage that we need to instrument is probably much higher for statement coverage, but this shows us that it is possible to reduce the number of statements instrumented to reduce the impact of the coverage without reducing the quality of the coverage.

Testing

Adding tests is always a good idea in a project, especially when the whole surrounding and content of the code is devoted to change. One way to test the code is to compile, run the code, and then check the output of the statement data file and the trace file matches the expected values. The current version of the code does not implement automated testing because the implementation will change soon to a compiler plug-in. Still, during the implementation, I made sure to keep all the code samples that I wrote to test the different situations that can happen in the code. These different code samples can be provisory checked manually, and are ready to be used in an automated way if a plug-in eventually comes up.

Conclusion

The goal is reached, we currently have a coverage tool that works on Dotty and have successfully avoided some of the misleading output that we can find in other coverage tools. We have seen that this version of statement coverage provides useful information for the user, but this seems to be achieved with some non-negligible costs.

Discovering the Dotty project was really interesting and at the same time really challenging, as the environment is complex and constantly evolves. Even if, at first glance, we might think that this work is completely separated from the compiler, there is in fact a lot of interactions with the surrounding. A first

example is some positions that were not correctly set during the other phases of the compilation, resulting in erroneous results when we looked at the report. To solve this problem, I had to go through the different transformations that happen to the tree to try to identify where the position was incorrectly set. A second example is the lifting of function arguments were part of the logic is reused from Eta-expansion. This work has taught me how to face an important code base, competence that will definitely be useful in my future.

Future work

Despite the fact that the current implementation works and produces a result with a decent amount of overhead, it is far from being ready. The code could be adapted to a separated compiler plug-in, thoroughly tested, debugged to support large code instrumentation, optimized, tested on other larger code like Dotty itself and much more other amelioration. This subject is in fact the door to a lot of interesting work that touch a lot of different domains. This report has introduced some challenge that remains to be solved, with some clue where to start to solve them.

Thanks

Professor Martin Odersky for supervising my project.

Guillaume Martres for his help and advices during the whole project.

Olivier Blanvillain for his advices for the different benchmarks used for this project.

References

[1] "SCoverage, code coverage tool for scala," [Online]. Available: http://www.scoverage.org/.

[2] *Chilakamarri, Kalyan-Ram & Elbaum, S. (2004). Reducing coverage collection overhead with disposable instrumentation. 233- 244. 10.1109/ISSRE.2004.32..*

[3] "Github repository : Scala Community Bench," [Online]. Available: https://github.com/OlivierBlanvillain/scala-community-bench.

[4] *A. Orso, A. Rao and M. J. Harrold, "A technique for dynamic updating of Java software," International Conference on Software Maintenance, 2002. Proceedings., 2002, pp. 649-658.*

[5] *H. Agrawal. Dominators, Super Blocks, and Program Coverage. In Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '94).*

[6] *J. Bowring, A. Orso, and M. Harrold. Monitoring Deployed Software Using Software Tomography. In Workshop on Program analysis for software tools and engineering.*

Source code

Dotty Coverage tool

https://github.com/Qjaquier/dotty-coverage-tools

Example of project that uses the version of the Dotty compiler that implements the coverage phase. It contains all the infrastructure to compile and run instrumented code, and generate a coverage report for it.

Dotty with coverage

https://github.com/Qjaquier/dotty/tree/coverage

Fork of dotty, implementing the coverage phase.