Joint R&D Project

Collaborative Applied Research and Development between

Morgan Stanley and SJTU

Project Name	Performance Enhancement of Scala Source Code
Project Leader	Prof. Kenny Zhu
Project Members	Yi Liu, Yu Shi, Pei Lv, Benxuan Zhang
Lab Name	Advance Data and Programming Technology, SJTU

Performance Enhancement of Scala Source Code

Abstract

Scala is one of the most popular programming languages for large scale real world applications. The heavy cost of compiling time makes developer waste a lot of time to just wait the build of projects. It is speculated that there exists a fair amount of “dead code” in Scala compiler code base, which adversely affects its performance.

In this project, we have a dead code detection tool such as Scala plugin to detect unnecessary code in Scala. Every time the Scala compiler compiles a program, the tool detects code that is not executed. Meanwhile, it records the data flow of each execution and thus identifies code which was executed but having no effect on the results. For the potential dead codes found by the tool, we build our own human analysis model to check whether they are real dead codes. For the first type of code, we do Call Trace Analysis. This is the most useful method to judge whether the dead code is true. At the same time, we analysis the function of such codes to judge if it is useful in code base. Our judgements include tree types of levels: file level, block level and sentence level. Analyzing codes by different level help us to find dead codes accurately and efficiently. For the second and third type of dead code, we check the function of potential codes given by the tool to determine the true dead code.

After finding the dead code inside the Scala compiler code, we submit our analysis results to Typesafe, so that they can improve their compiler codes. Also, we send our feedback for the false deadcode to the detection tool owner.

Contents

1. Introduction

Scala programs are widely used on Morgan Stanley's distributed data processing and computing platforms. However, the compilation process of these programs takes typically 3-4 times longer than equivalent Java programs. This relatively poor performance leads to substantial negative effect on Morgan Stanley business. The key technical challenge of this project is to analyze the source codes of Scala compiler and do human check for report of the dead codes identified by dynamic detection tool.

1.1 Problem Background

The Scala deadcode detection tool gives the potential deadcode during Scala compilation. We need to look through the report and find their position in source code base. All the possible dead codes in source codes must be analyzed based on its type, context code base or the function of the file it in. After analyzing the dead codes in the Scala compiler, we submit these findings to Typesafe to give them suggestions to enhance the Scala 2.12 compiler. The improvement of Scala compiler will bring large benefit to Morgan Stanley technical team as well as Scala community.

1.2 Analysis of the Problem

Since all the dead codes are divided into three types, we build a framework for our approach. We first preprocess the dead codes and extract the executed trace. We then analyze dead codes given by the tool with its adjacent codes and how these adjacent codes execute. The context information of so called dead code allows us to understand how these codes executed and the function of this codes the developer of the Scala compiler want to give. For different function of codes, we will test them by compiling many Scala programs in come extreme circumstances to judge the unexecuted. Code that has never been reached and has nothing to do with the compiler's performance will be type 1 dead code.

For type 2 dead code detection, the existing tool will produce a file of data flow during compilation in order to find useless data dependencies which doesn't produce meaningful output. After data dependency analyzing, the tool will show how the codes which are executable and have no effect to the final results. We need to check the call history of these codes step by step and find whether these codes will affect the results in some extreme situation.

For type 3 dead code, we will judge whether there is I/O execution. After ensuring the real dead code inside Scala compiler, we will create pull request to Scala GitHub or submit requests to Typesafe Jenkins system.

2. Types of Potential Dead code and Analysis Method

The basic idea of the tool we use is to transform the AST of Scala compiler source by adding "println" expression in front of every executable line. In order to track the behavior of Scala compiler, the tool not only outputs codes but also outputs the location of every executed line. The tool transforms the whole AST of a source file and each node of which denotes a construct occurring in the source code. Exhaustive testing by compiling many different Scala programs allows the tool to explore the execution space in the compiler. Code that has never been reached can be marked as potential type 1 dead code. Code that has been reached but has nothing to do with the final result due to data dependency analysis can be marked as type 2 or 3 dead codes.

In order to find real dead codes, we create a methodology to analyze the result and executed trace from the tool. Firstly, we preprocess the dead code and executed trace. Then, we divide the dead codes in three types and handle them separately. We test and verify the compiler when we delete the code seen as dead code. After human checking, we send the report of wrong dead codes for dead code detection tool team and make suggestions to improve their tool. We submit the analysis of real deadcode Typesafe. The Figure 1. Shows the framework of how to do human analysis.

2.1 Preprocess

Before human analysis, we need to look at the data flow of the detection tool. In this way, we can understand the executed process of the statements in Scala compiler. Then, we check the format of the results given by the previous deadcode detection tool.

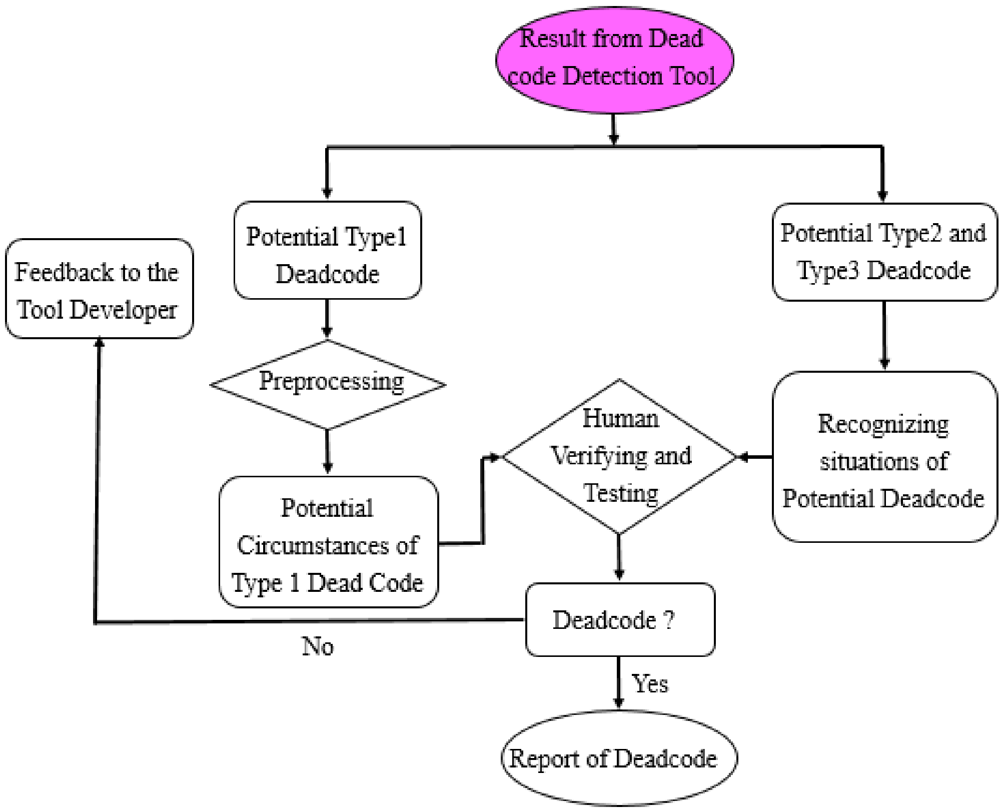
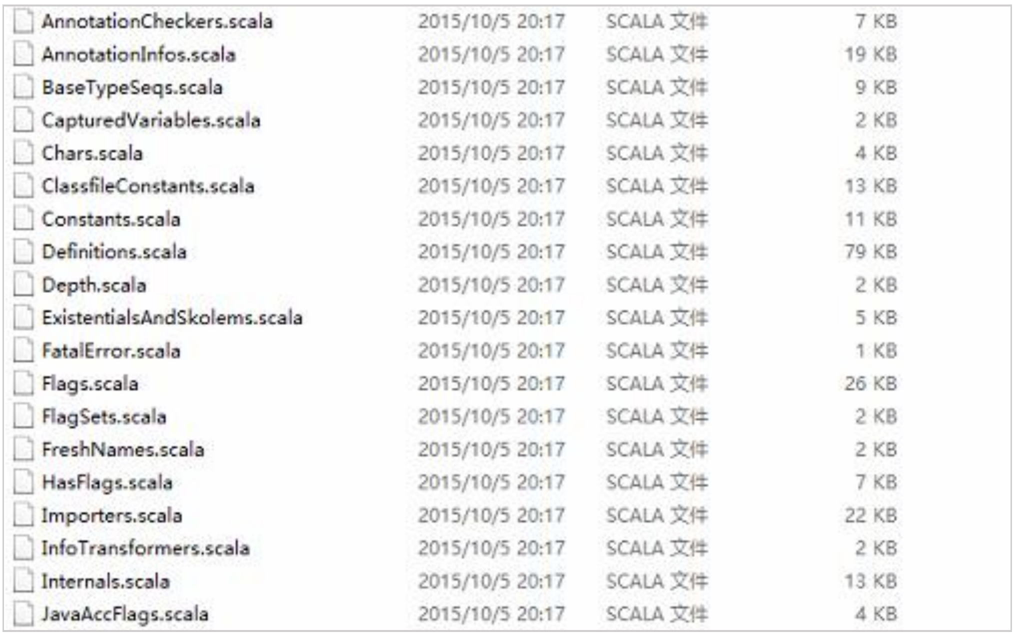

Result from Dead code Detection Tool
Potential Type1 Deadcode
Potential Type2 and Type3 Deadcode
Feedback to the Tool Developer
Preprocessing
Human Verifying and Testing
Recognizing situations of Potential Deadcode
Potential Circumstances of Type 1 Dead Code
Deadcode ?
No
Yes
Report of Deadcode

Figure 1. Our Framework of Human Analysis

2.2 Potential Circumstances of Type 1 Dead Code

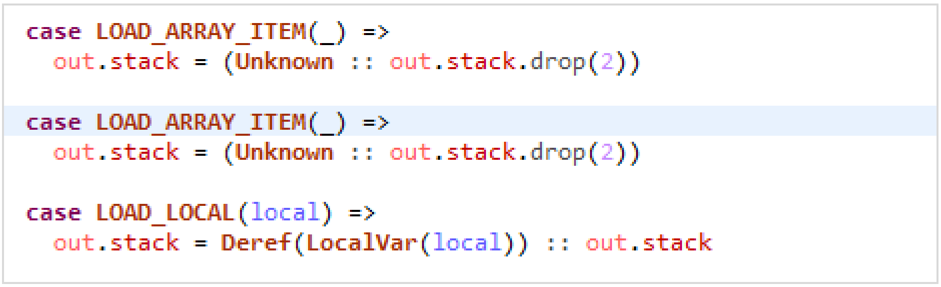
Code Detection tool can output files with all possible dead codes and their positions in Scala compiler. We divide type 1 dead code in these six situations. The first situation is about the whole Scala file which is never used during the whole compilation. These kind of dead codes are show in Figure 2.



AnnotationCheckers.scala	2015/10/5 20:17	SCALA 文件	7 KB
AnnotationInfos.scala	2015/10/5 20:17	SCALA 文件	19 KB
BaseTypeSeqs.scala	2015/10/5 20:17	SCALA 文件	9 KB
CapturedVariables.scala	2015/10/5 20:17	SCALA 文件	2 KB
Chars.scala	2015/10/5 20:17	SCALA 文件	4 KB
ClassfileConstants.scala	2015/10/5 20:17	SCALA 文件	13 KB
Constants.scala	2015/10/5 20:17	SCALA 文件	11 KB
Definitions.scala	2015/10/5 20:17	SCALA 文件	79 KB
Depth.scala	2015/10/5 20:17	SCALA 文件	2 KB
ExistentialsAndSkolems.scala	2015/10/5 20:17	SCALA 文件	5 KB
FatalError.scala	2015/10/5 20:17	SCALA 文件	1 KB
Flags.scala	2015/10/5 20:17	SCALA 文件	26 KB
FlagSets.scala	2015/10/5 20:17	SCALA 文件	2 KB
FreshNames.scala	2015/10/5 20:17	SCALA 文件	2 KB
HasFlags.scala	2015/10/5 20:17	SCALA 文件	7 KB
Importers.scala	2015/10/5 20:17	SCALA 文件	22 KB
InfoTransformers.scala	2015/10/5 20:17	SCALA 文件	2 KB
Internals.scala	2015/10/5 20:17	SCALA 文件	13 KB
JavaAccFlags.scala	2015/10/5 20:17	SCALA 文件	4 KB

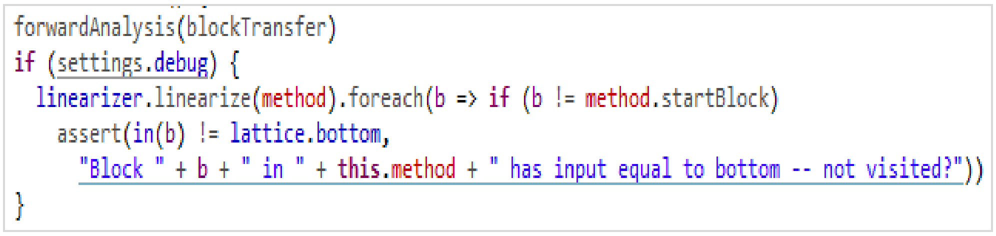
Figure 2. Whole file potentially never used in Scala compiler

The second goes with the dead code which is used for debugging but left over by developer. For example, the developer may build case or if condition statement for the compiler, but the condition statement will never happen during the compilation process. The following Figure 3 and Figure 4 give the example of these two circumstances.



```
case LOAD_ARRAY_ITEM(_) =>
  out.stack = (Unknown :: out.stack.drop(2))

case LOAD_ARRAY_ITEM(_) =>
  out.stack = (Unknown :: out.stack.drop(2))

case LOAD_LOCAL(local) =>
  out.stack = Deref(LocalVar(local)) :: out.stack
```

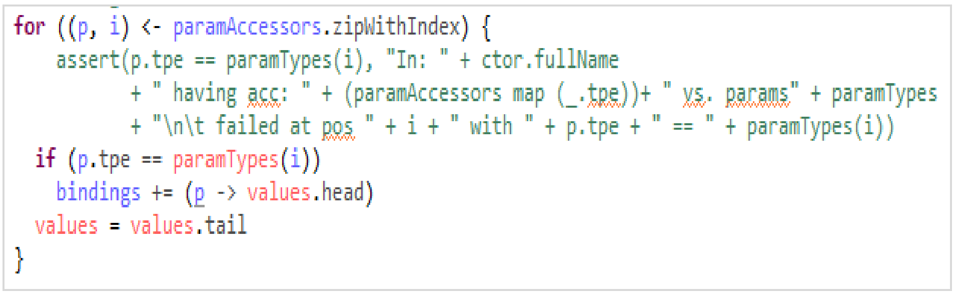
Figure 3. Judging Case Statement in Scala compiler



```
forwardAnalysis(blockTransfer)
if (settings.debug) {
  linearizer.linearize(method).foreach(b => if (b != method.startBlock)
    assert(in(b) != lattice.bottom,
      "Block " + b + " in " + this.method + " has input equal to bottom -- not visited?"))
}
```

Figure 4. Judging If Statement in Scala Compiler

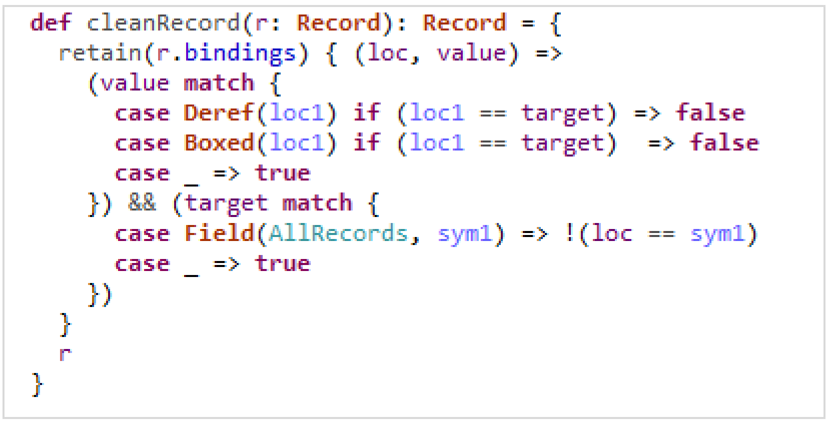
Figure 3 shows that bodies of two neighboring cases are identical and could be merged. After human analysis, the second case statement can be moved. There are also another circumstances for case statement like the condition of the statement will never happen. We need to check them step by step. Figure 4 shows the possible dead code structure of if statement, if the condition is always false then the following statements in Scala compiler can be seen as real dead code, otherwise they are not dead codes.



```
for ((p, i) <- paramAccessors.zipWithIndex) {
    assert(p.tpe == paramTypes(i), "In: " + ctor.fullName
          + " having acc: " + (paramAccessors map (_.tpe))+ " vs. params" + paramTypes
          + "\n\t failed at pos " + i + " with " + p.tpe + " == " + paramTypes(i))
  if (p.tpe == paramTypes(i))
    bindings += (p -> values.head)
  values = values.tail
}
```

Figure 5. Example of for Loop in Scala compiler

There also will be for loop or while loop that never reach the executable conditions, just like statements shown in Figure 5. We need to do human judgement about whether the conditions will be true. The third situation is that the function in the class that will be never used. The human analysis will check the possible call history and call path of this function.



```
def cleanRecord(r: Record): Record = {
  retain(r.bindings) { (loc, value) =>
    (value match {
      case Deref(loc1) if (loc1 == target) => false
      case Boxed(loc1) if (loc1 == target)  => false
      case _ => true
    }) && (target match {
      case Field(AllRecords, sym1) => !(loc == sym1)
      case _ => true
    })
  }
  r
}
```

Figure 6. Example of function in Scala compiler

The fourth situation is about the assignment statement and variable declaration. In human analysis, we will check what these variables used for and unused variables will be moved. The final situation is to deal with the source files which is never touched during evaluation. For the final situation, we need to read the notes about the related files and understand their functions.

We first do human check and clearly knowing the pattern of different types of dead codes, then develop our analysis technological process to recognize the dead codes in different situations. In this way, we will go through all the type 1 dead codes found in the tool and get the mostly possible dead codes by human checking.

2.3 Type 2 Dead Code Verifying

Since type 2 dead code is detected by tool using data dependency, we do human analysis by the following step. First, we scan the whole list of results and judge whether there is wrong output of the detection tool. Second, we focus on the dead code in the position of files and read the context of full content. Then, we divided them into different patterns of potential dead codes. Also, we extract the read and write statements related to dead codes. We investigate whether the statement has nothing to do with the final results in most cases, but there are some special circumstances it affects the final results. We also check the trace given by the type 2 deadcode detection tool.

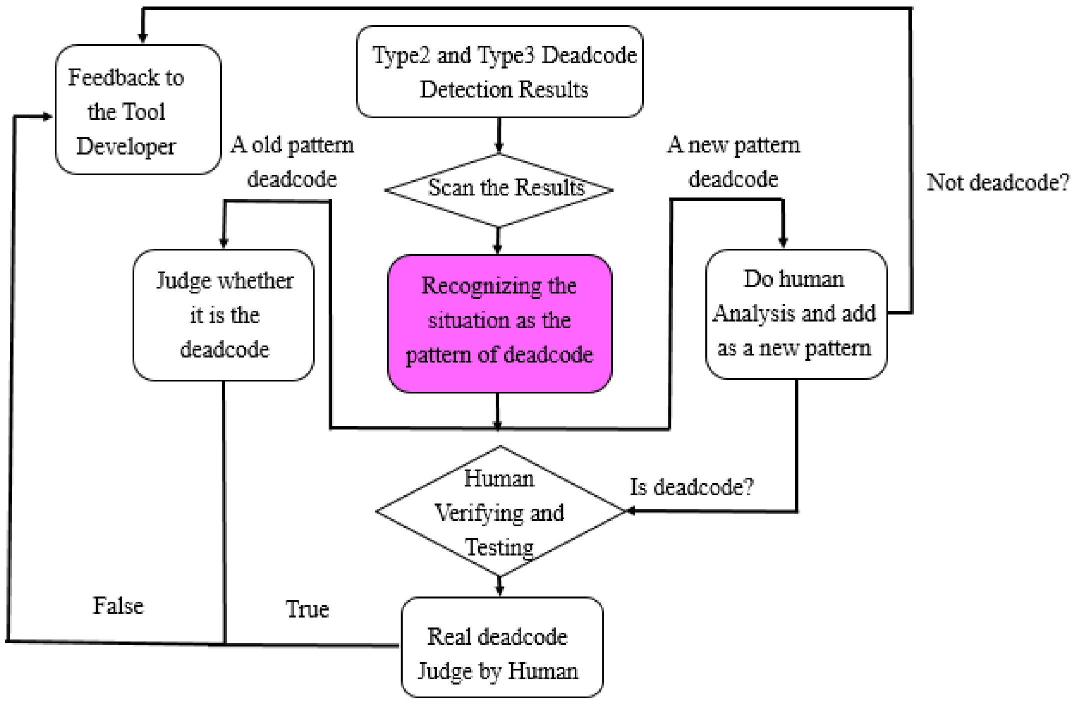

Feedback to the Tool Developer
Type2 and Type3 Deadcode Detection Results
A old pattern deadcode
Scan the Results
A new pattern deadcode
Not deadcode?
Judge whether it is the deadcode
Recognizing the situation as the pattern of deadcode
Do human Analysis and add as a new pattern
Human Verifying and Testing
Is deadcode?
False
True
Real deadcode Judge by Human

Figure 7. Framework for Type 2 Dead Code Verifying

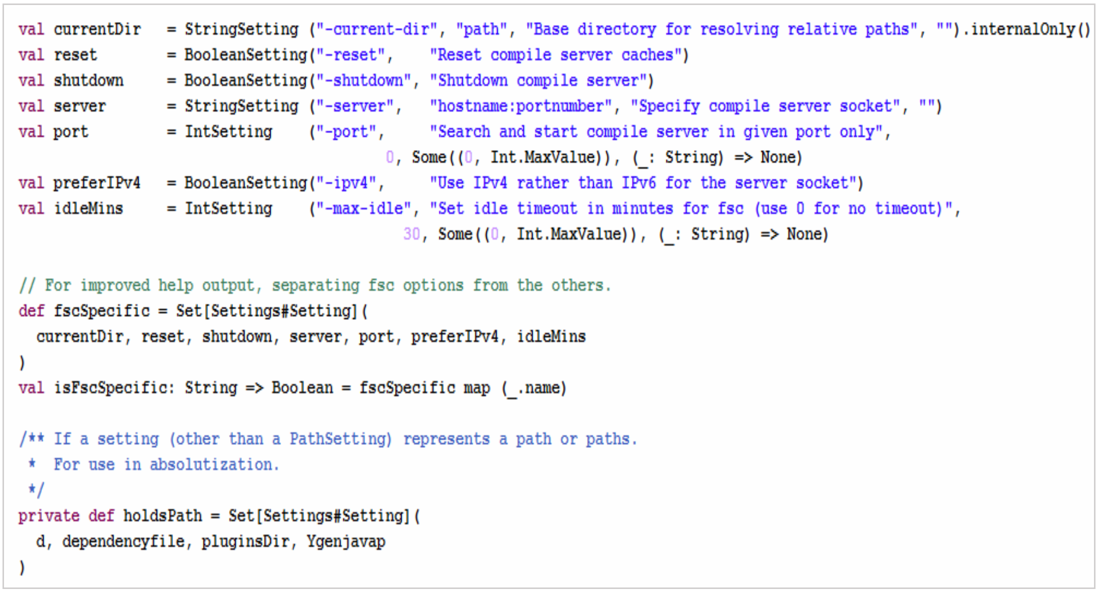
As shown in Figure 7, we follow the results given by the detection tool and judge the dead code by human. If the code is real code, we will add its pattern to one of our judgement rules, otherwise we will write a feedback to the developer of dead code detection tool.

2.4 Type 3 Dead Code Verfying

The detection of Type 3 dead code is based on Type 2. To find out I/O operations during compiling, the tool just need to search the dead codes in Type 2 and pick out those with names of I/O methods and objects, such as “print”, “println”, “Source” and other names in scala.io package. Also, the tool should look for basic I/O function names defined in java, such as “java.io.FileWriter”, “write”, etc. So during human analysis, we will judge whether there exists I/O operation. Also we need to change the value of variables to check if there’s an exception.

3. Analysis Procedure and Results

After well designed our human checking method about three types of dead codes, we still need to do one step that is verifying and Testing. Our approach is delete a batch of codes with the same pattern and see whether it affects the performance of Scala compiler. We use scalac, fsc and different options in scala compiler to test the potential dead codes. After using different options, we find some potential deadcode detected by the deadcode detection tool is false. As it shown in Figure 8, the codes related to fsc options are seen as deadcode. We give our feedback to the tool developers and so they can improve their tool.



```
val currentDir   = StringSetting ("-current-dir", "path", "Base directory for resolving relative paths", "").internalOnly()
val reset        = BooleanSetting("-reset",    "Reset compile server caches")
val shutdown     = BooleanSetting("-shutdown", "Shutdown compile server")
val server       = StringSetting ("-server",   "hostname:portnumber", "Specify compile server socket", "")
val port         = IntSetting    ("-port",     "Search and start compile server in given port only",
                                   0, Some((0, Int.MaxValue)), (_: String) => None)
val preferIPv4   = BooleanSetting("-ipv4",     "Use IPv4 rather than IPv6 for the server socket")
val idleMins     = IntSetting    ("-max-idle", "Set idle timeout in minutes for fsc (use 0 for no timeout)",
                                   30, Some((0, Int.MaxValue)), (_: String) => None)

// For improved help output, separating fsc options from the others.
def fscSpecific = Set[Settings#Setting](
  currentDir, reset, shutdown, server, port, preferIPv4, idleMins
)
val isFscSpecific: String => Boolean = fscSpecific map (_.name)

/** If a setting (other than a PathSetting) represents a path or paths.
 *  For use in absolutization.
 */
private def holdsPath = Set[Settings#Setting](
  d, dependencyfile, pluginsDir, Ygenjavap
)
```

Figure 8. The codes are related to fsc options

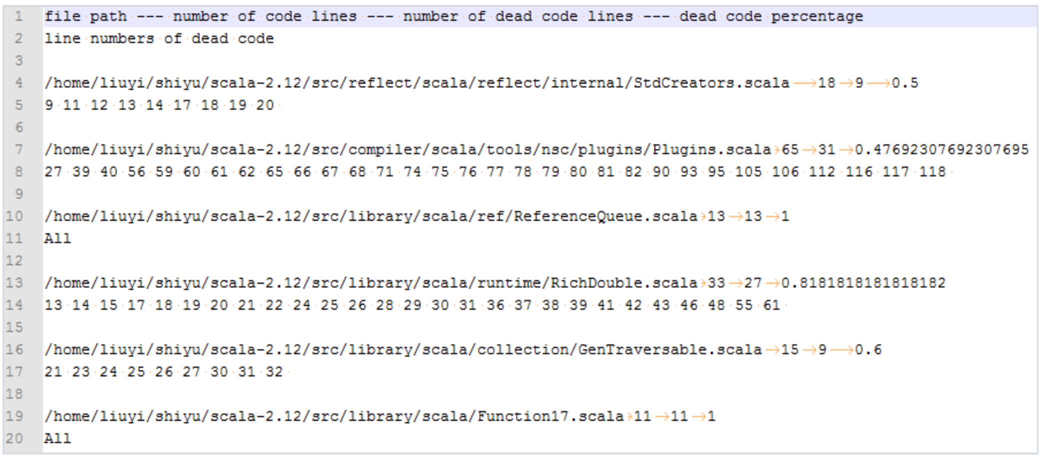
These kind of code are useful when we run in fsc mode, so all of code like this cannot be seen as type 1 deadcode. When we test codes in different options or different flags, these code can be touched. So we will give our analysis report to the tool developer to improve their tools and delete these false dead codes in their analysis results.

3.1 Output file of Potential deadcode of Type 1

By using the dynamic detecting tool of Scala compiler based on compiler plugin, we successfully found some first type of dead code in the compiler source. However, some of them may not be dead code anymore if we try more input program and do more test. In order to decrease the burden of test, we need to figure out some ways to verify the dead code manually.

Here we use an additional program to sum the source code together and give us statistics about the dead code. The program takes the output file of the compiler as the input and then it will output two files, one about the statistics and the other about the code itself for manual verification. The input file is simple, including the path of the original program and the current line numbers.

The first output file contains the lines which are treated as dead code and also the percentage of dead code. It looks in this way:

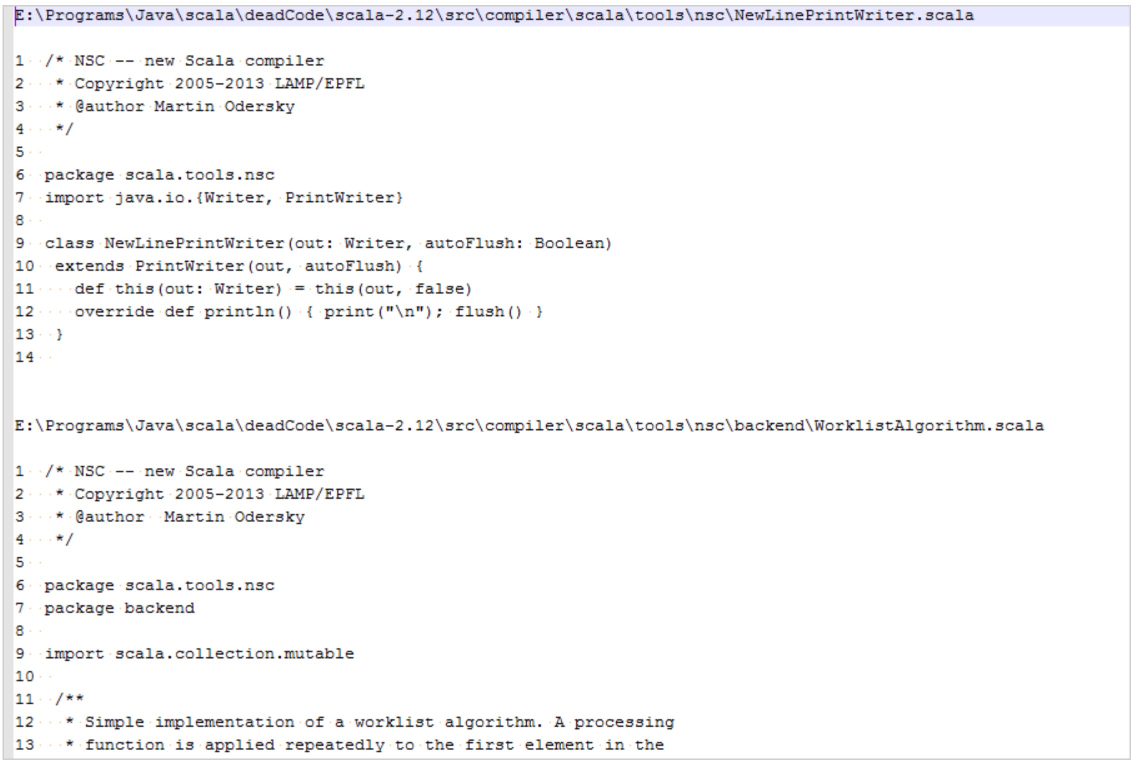


```
file path --- number of code lines --- number of dead code lines --- dead code percentage
line numbers of dead code

/home/liuyi/shiyu/scala-2.12/src/reflect/scala/reflect/internal/StdCreators.scala	18	9	0.5
9 11 12 13 14 17 18 19 20 

/home/liuyi/shiyu/scala-2.12/src/compiler/scala/tools/nsc/plugins/Plugins.scala	65	31	0.47692307692307695
27 39 40 56 59 60 61 62 65 66 67 68 71 74 75 76 77 78 79 80 81 82 90 93 95 105 106 112 116 117 118 

/home/liuyi/shiyu/scala-2.12/src/library/scala/ref/ReferenceQueue.scala	13	13	1
All

/home/liuyi/shiyu/scala-2.12/src/library/scala/runtime/RichDouble.scala	33	27	0.8181818181818182
13 14 15 17 18 19 20 21 22 24 25 26 28 29 30 31 36 37 38 39 41 42 43 46 48 55 61 

/home/liuyi/shiyu/scala-2.12/src/library/scala/collection/GenTraversable.scala	15	9	0.6
21 23 24 25 26 27 30 31 32 

/home/liuyi/shiyu/scala-2.12/src/library/scala/Function17.scala	11	11	1
All
```

Figure 9. The output of Dynamic Analysis Tool

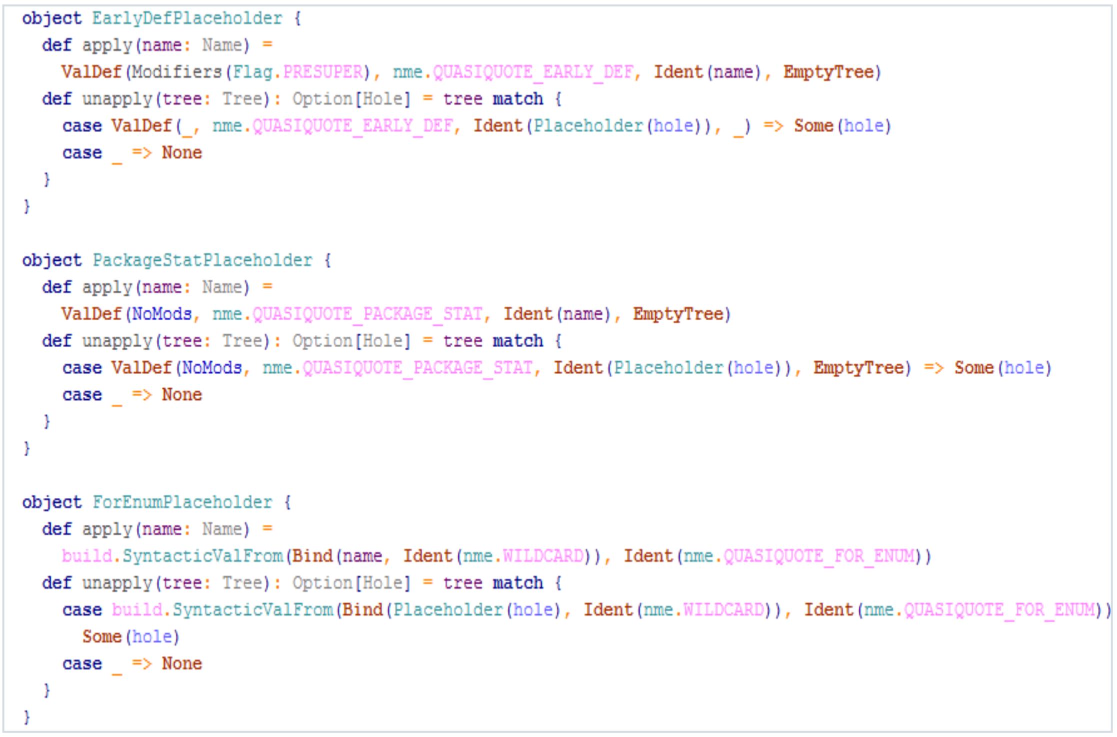
Here we can see that in some files, all the codes are considered dead and in the other files, there's a trace for dead code.

The second program reads the source files and then show us the contents of the files with dead code high-lighted. Also in this program we add some additional logic to ensure that only functional codes are included dead, which means lines containing "import", "{" and "}" are excluded. Then the output file looks in this way, in which all non-dead codes are in comment. The first is an example of file in which all codes are considered dead here by compiler plugin tool.



```
E:\Programs\Java\scala\deadCode\scala-2.12\src\compiler\scala\tools\nsc\NewLinePrintWriter.scala

/* NSC -- new Scala compiler
  * Copyright 2005-2013 LAMP/EPFL
  * @author Martin Odersky
  */

package scala.tools.nsc
import java.io.{Writer, PrintWriter}

class NewLinePrintWriter(out: Writer, autoFlush: Boolean)
 extends PrintWriter(out, autoFlush) {
   def this(out: Writer) = this(out, false)
   override def println() { print("\n"); flush() }
 }


E:\Programs\Java\scala\deadCode\scala-2.12\src\compiler\scala\tools\nsc\backend\WorklistAlgorithm.scala

/* NSC -- new Scala compiler
  * Copyright 2005-2013 LAMP/EPFL
  * @author  Martin Odersky
  */

package scala.tools.nsc
package backend

import scala.collection.mutable

 /**
   * Simple implementation of a worklist algorithm. A processing
   * function is applied repeatedly to the first element in the
```

Figure 10. The Example of file contents codes which are all deadcode



```
object EarlyDefPlaceholder {
  def apply(name: Name) =
    ValDef(Modifiers(Flag.PRESUPER), nme.QUASIQUOTE_EARLY_DEF, Ident(name), EmptyTree)
  def unapply(tree: Tree): Option[Hole] = tree match {
    case ValDef(_, nme.QUASIQUOTE_EARLY_DEF, Ident(Placeholder(hole)), _) => Some(hole)
    case _ => None
  }
}

object PackageStatPlaceholder {
  def apply(name: Name) =
    ValDef(NoMods, nme.QUASIQUOTE_PACKAGE_STAT, Ident(name), EmptyTree)
  def unapply(tree: Tree): Option[Hole] = tree match {
    case ValDef(NoMods, nme.QUASIQUOTE_PACKAGE_STAT, Ident(Placeholder(hole)), EmptyTree) => Some(hole)
    case _ => None
  }
}

object ForEnumPlaceholder {
  def apply(name: Name) =
    build.SyntacticValFrom(Bind(name, Ident(nme.WILDCARD)), Ident(nme.QUASIQUOTE_FOR_ENUM))
  def unapply(tree: Tree): Option[Hole] = tree match {
    case build.SyntacticValFrom(Bind(Placeholder(hole), Ident(nme.WILDCARD)), Ident(nme.QUASIQUOTE_FOR_ENUM))
      Some(hole)
    case _ => None
  }
}
```

Figure 11. All codes are all useless in a file

With the help of these tools, we could easily get all suspicious dead codes and begin to verify manually.

From the output file of the tools, we find that about 60% code of the compiler source code are considered dead, according to the current inputs. And almost all of them are in blocks, which means many lines of code are dead or not dead simultaneously. So we will discuss the dead code in the unit of block later.

3.2 Call Trace Analysis

This method is the most useful one and we could deal with more than 70% of suspicious dead code using it. Also, by using it we could straightly get conclusions about whether it is dead code block.

Call Trace Analysis works in the same way as the referencing counting in garbage collection. For a block of code, we need to find its caller and its caller's caller. By using the IDE, we could easily find the caller of a function or class. Here we need to discuss three different situations.

First, if a block has no caller or the caller has no caller, we could be sure that the whole calling trace is not called under all circumstances. And then these blocks are surely dead code.

Second, if a block has many callers and all of them are functions or methods, then we should first decide whether all of them are dead code. If any one of them are not, then the block itself is also not dead code. But if all of them are, it will also be dead code. The way to decide whether they are dead code is same because functions and classes must have a caller, too.

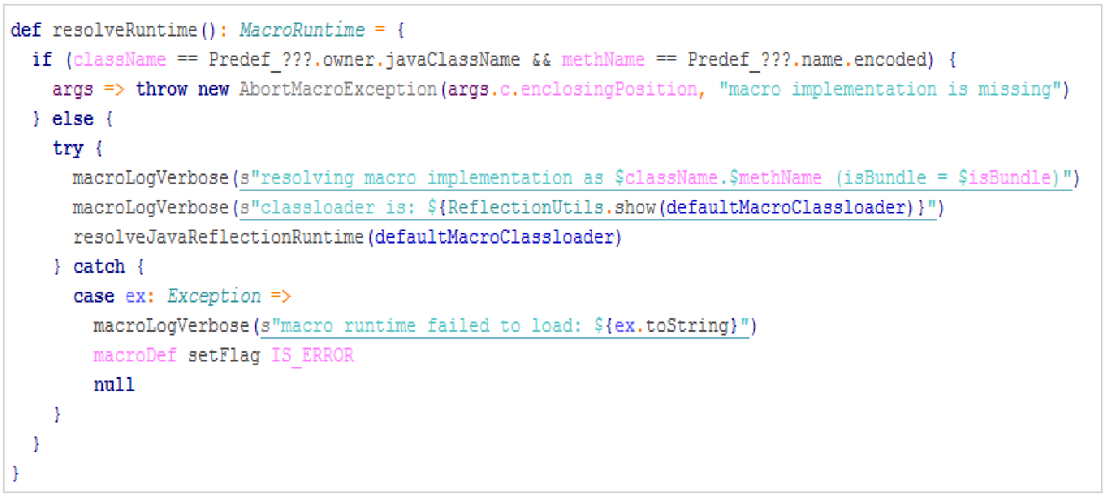
Third, if a block's callers contain blocks within "if" or "case", it will be much difficult to judge. Since "if" and "match" always have more than one traces and they may run different ways according to different methods, we couldn't verify it statically and only more tests could solve the problem. Here's an example of the call trace, we note the caller as source file package or line number. Some of if condition sentences ever occurred like the following example.

So we need the method in the follow section, which introduces how to analysis the Keywords in Scala compiler.

3.2 Keywords Analysis

Catch

For some blocks, they have some keywords, indicating the basic function of the block. So during the verification we could first find the keywords to judge the block. The most useful one of the keywords is "catch".



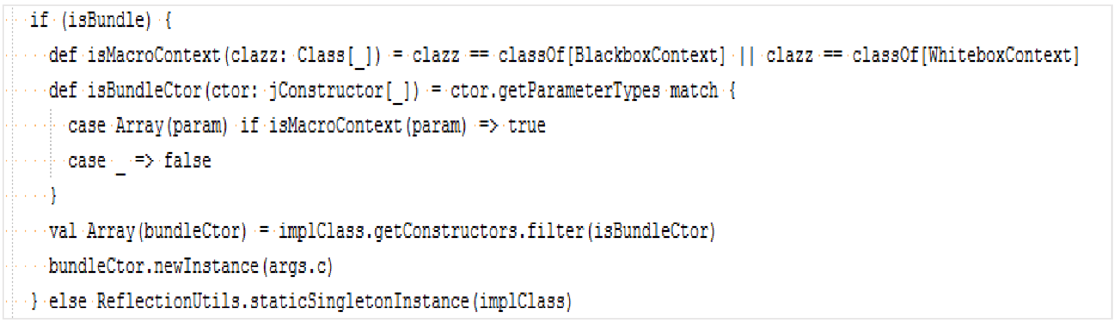
```
def resolveRuntime(): MacroRuntime = {
  if (className == Predef_???.owner.javaClassName && methName == Predef_???.name.encoded) {
    args => throw new AbortMacroException(args.c.enclosingPosition, "macro implementation is missing")
  } else {
    try {
      macroLogVerbose(s"resolving macro implementation as $className.$methName (isBundle = $isBundle)")
      macroLogVerbose(s"classloader is: ${ReflectionUtils.show(defaultMacroClassloader)}")
      resolveJavaReflectionRuntime(defaultMacroClassloader)
    } catch {
      case ex: Exception =>
        macroLogVerbose(s"macro runtime failed to load: ${ex.toString}")
        macroDef setFlag IS_ERROR
        null
    }
  }
}
```

Figure 12. The Keywords "Catch"

There're many blocks like this. The block in "catch" is not executed and is considered dead code. However, most time if we add a program including some exception, the block could be executed. To avoid such test, we could simply think that blocks within "catch" are not dead code.

If and Else

When dealing with If-Else condition situation, we can hardly know whether the branch of the statement really happened. We do two things to judge the situation. First, we add more test case to see whether the code can be touched. Second, we analysis the function of this block to judge whether the function is necessary.



```
if (isBundle) {
  def isMacroContext(clazz: Class[_]) = clazz == classOf[BlackboxContext] || clazz == classOf[WhiteboxContext]
  def isBundleCtor(ctor: jConstructor[_]) = ctor.getParameterTypes match {
    case Array(param) if isMacroContext(param) => true
    case _ => false
  }
  val Array(bundleCtor) = implClass.getConstructors.filter(isBundleCtor)
  bundleCtor.newInstance(args.c)
} else ReflectionUtils.staticSingletonInstance(implClass)
```

Figure 13. If and Else Condition Situation

In figure 13, the else part is never executed, so we can remove it.

Case

Judging the keyword ‘case’ of pattern match is also a difficult task. The most important thing to dealing with this kind of dead codes is to find the original statement which we need to match and analysis the potential cases it will have.

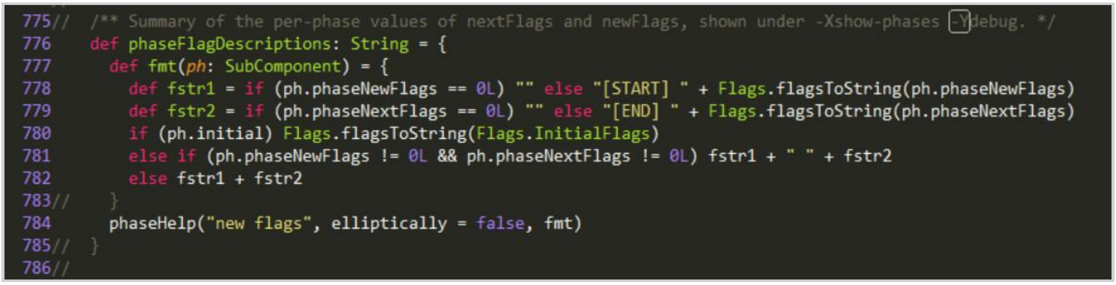
For and While Loop

For this kind of situation, we check the condition that makes loop happen and find all the condition are ok to be the original condition to make the loop happen.

Function Block
Some of the function block are useless, because they not used in the following execution of compiler. They are just leftover by developer. We can remove these function block and it has no influence on our execution of codes.

3.4 Test Cases

For some blocks or files, they have no callers in the source code but they could be executed during the compiling process. We guess they are called by other binary code or the JVM during the compiling. Luckily this kind of code are rare and they all have comments showing us when they could be called. For example the command used to show phases must be called under some option.



```
  /** Summary of the per-phase values of nextFlags and newFlags, shown under -Xshow-phases -Ydebug. */
  def phaseFlagDescriptions: String = {
    def fmt(ph: SubComponent) = {
      def fstr1 = if (ph.phaseNewFlags == 0L) "" else "[START] " + Flags.flagsToString(ph.phaseNewFlags)
      def fstr2 = if (ph.phaseNextFlags == 0L) "" else "[END] " + Flags.flagsToString(ph.phaseNextFlags)
      if (ph.initial) Flags.flagsToString(Flags.InitialFlags)
      else if (ph.phaseNewFlags != 0L && ph.phaseNextFlags != 0L) fstr1 + " " + fstr2
      else fstr1 + fstr2
    }
    phaseHelp("new flags", elliptically = false, fmt)
  }
```

Figure 14. Blocks or Files with no Callers in the Source Code

For these kind of code, we need to design some test cases and run them under special options. Then if we find that some blocks are executed, we could exclude them from dead code.

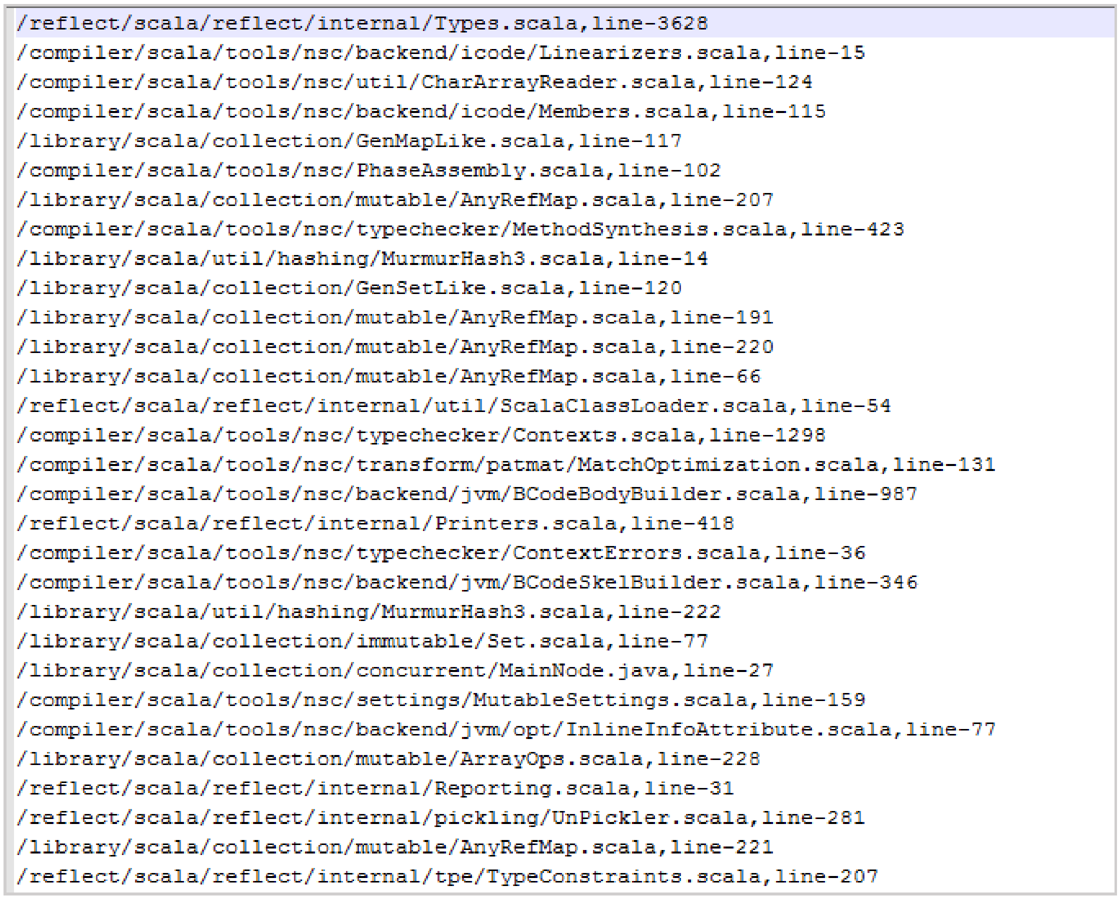
For some special blocks, if the previous three ways could not work normally, we then must use this method to verify the dead code. According to the definition of type one dead code, a dead code block will never be called. So for a block, we find all its call trace and then remove them all from the compiler source. Then we re-compiler the compiler and run it and see the result. If the compiler run normally and the results are all the same with previous one, we then could decide that the block is dead code.

By using the methods mentioned above, we have successfully found some dead codes and they can be categorized into three types. Here're the dead code we have found after verification. The whole classes are dead code.

3.5 The Result of Type 2 and Type 3 dead code

Through our experiments, we can conclude that compiler plugin with Icode approach is feasible avenue for source-level dynamic dead code detection. The plugin with AST approach has relatively poor performance. This time we use the result from Icode to as our main reference and compare them with the result from plugin with AST approach. Type 2 and 3 dead code has been successfully detected in the scala source. Since Type 3 is included in Type 2 dead code, so when we analysis type 2 dead code we analysis type 3 dead code as well.

During our test we find the following potential dead codes.



```
/reflect/scala/reflect/internal/Types.scala,line-3628
/compiler/scala/tools/nsc/backend/icode/Linearizers.scala,line-15
/compiler/scala/tools/nsc/util/CharArrayReader.scala,line-124
/compiler/scala/tools/nsc/backend/icode/Members.scala,line-115
/library/scala/collection/GenMapLike.scala,line-117
/compiler/scala/tools/nsc/PhaseAssembly.scala,line-102
/library/scala/collection/mutable/AnyRefMap.scala,line-207
/compiler/scala/tools/nsc/typechecker/MethodSynthesis.scala,line-423
/library/scala/util/hashing/MurmurHash3.scala,line-14
/library/scala/collection/GenSetLike.scala,line-120
/library/scala/collection/mutable/AnyRefMap.scala,line-191
/library/scala/collection/mutable/AnyRefMap.scala,line-220
/library/scala/collection/mutable/AnyRefMap.scala,line-66
/reflect/scala/reflect/internal/util/ScalaClassLoader.scala,line-54
/compiler/scala/tools/nsc/typechecker/Contexts.scala,line-1298
/compiler/scala/tools/nsc/transform/patmat/MatchOptimization.scala,line-131
/compiler/scala/tools/nsc/backend/jvm/BCodeBodyBuilder.scala,line-987
/reflect/scala/reflect/internal/Printers.scala,line-418
/compiler/scala/tools/nsc/typechecker/ContextErrors.scala,line-36
/compiler/scala/tools/nsc/backend/jvm/BCodeSkelBuilder.scala,line-346
/library/scala/util/hashing/MurmurHash3.scala,line-222
/library/scala/collection/immutable/Set.scala,line-77
/library/scala/collection/concurrent/MainNode.java,line-27
/compiler/scala/tools/nsc/settings/MutableSettings.scala,line-159
/compiler/scala/tools/nsc/backend/jvm/opt/InlineInfoAttribute.scala,line-77
/library/scala/collection/mutable/ArrayOps.scala,line-228
/reflect/scala/reflect/internal/Reporting.scala,line-31
/reflect/scala/reflect/internal/pickling/UnPickler.scala,line-281
/library/scala/collection/mutable/AnyRefMap.scala,line-221
/reflect/scala/reflect/internal/tpe/TypeConstraints.scala,line-207
```

Figure 15. Results Generated by Type 2 Deadcode

Real Deadcode Detected by Human

We analysis the codes step by step. For example, there are dead codes in scala compiler source codes.

The code ***case _: DefTree => change(tree.symbol.moduleClass)*** is dead code, because it has no data dependency and has no external impacts. This function only change the internal members in Tree but will not infect the results, so we judge these codes are real dead codes.

```
override def traverse(tree: Tree) {
            tree match {
              case _: DefTree => change(tree.symbol.moduleClass)
              case _             =>
            }
            super.traverse(tree)
          }
```

we find some methods in compiler are not depend on global and do not have effect on the final results. For instance, we find such thread-safe method that depends only on the BTypes component, which does not depend on global.

```
    override def getCommonSuperClass(inameA: String, inameB: String): String = {
      val a = classBTypeFromInternalName(inameA)
      val b = classBTypeFromInternalName(inameB)
      val lub = a.jvmWiseLUB(b).get
```

```
          val lubName = lub.internalName
          assert(lubName != "scala/Any")
          lubName // ASM caches the answer during the lifetime of a ClassWriter. We
outlive that. Not sure whether caching on our side would improve things.
      }
```

Another Type 2 deadcode found by the tool,

```
override def write(cw: ClassWriter, code: Array[Byte], len: Int, maxStack: Int,
maxLocals: Int): ByteVector = {
      val result = new ByteVector()

      result.putByte(InlineInfoAttribute.VERSION)

      var finalSelfSam = 0
      if (inlineInfo.isEffectivelyFinal)                         finalSelfSam |= 1
      if (inlineInfo.traitImplClassSelfType.isDefined) finalSelfSam |= 2
      if (inlineInfo.sam.isDefined)                                finalSelfSam |= 4
      result.putByte(finalSelfSam)

      for (selfInternalName <- inlineInfo.traitImplClassSelfType) {
          result.putShort(cw.newUTF8(selfInternalName))
      }

      for (samNameDesc <- inlineInfo.sam) {
          val (name, desc) = samNameDesc.span(_ != '(')
          result.putShort(cw.newUTF8(name))
          result.putShort(cw.newUTF8(desc))
      }

      // The method count fits in a short (the methods_count in a classfile is also a
short)
      result.putShort(inlineInfo.methodInfos.size)

      // Sort the methodInfos for stability of classfiles
      for ((nameAndType, info) <- inlineInfo.methodInfos.toList.sortBy(_._1)) {
          val (name, desc) = nameAndType.span(_ != '(')
          // Name and desc are added separately because a NameAndType entry also
stores them separately.
          // This makes sure that we use the existing constant pool entries for the
method.
          result.putShort(cw.newUTF8(name))
          result.putShort(cw.newUTF8(desc))

          var inlineInfo = 0
          if (info.effectivelyFinal)                              inlineInfo |= 1
          if (info.traitMethodWithStaticImplementation) inlineInfo |= 2
          if (info.annotatedInline)                               inlineInfo |= 4
          if (info.annotatedNoInline)                             inlineInfo |= 8
          result.putByte(inlineInfo)
      }
```

```
    result
  }
```

This whole block is considered deadcode, because the return value of the function has no external impacts on the final result.

There are some codes or flags used for compiler developer to debug the codes, but it is leftover as codes which do not affect the execution of compiler. As it shown below,

```
okToCall = true // TODO: remove (debugging)
```

The *okToCall* can be remove, because this flag always true for debug use.

False Deadcode Judged by Human Analysis

On the other hand, some of the codes though have no external impacts but it optimize the compiling process. So we also see these codes as useful codes. For example,

```
/** Handle line ends */
  private def potentialLineEnd() {
    if (ch == LF || ch == FF) {
      lastLineStartOffset = lineStartOffset
      lineStartOffset = charOffset
    }
  }
```

These code define the function to handle line ends in scala compiler during compilation, so we cannot see it as dead code. We feed back these false dead codes by human analysis. Moreover, we find all the lines with function *def* is considered wrong detection of deadcode, so we tell this situation to Type 2 tool developer to improve their plugin.

Trait may not helpful for push/pop operation, but you can’t delete these kind of code:

```
trait Linearizers {
  self: ICodes =>

  import global.debuglog
  import opcodes._

  abstract class Linearizer {
    def linearize(c: IMethod): List[BasicBlock]
    def linearizeAt(c: IMethod, start: BasicBlock): List[BasicBlock]
  }
```

Also, there are some codes used for initialization and these codes can’t be deleted too. As it shown below,

```
      var lineInfo = 0
```

The *lineInfo* may not have impact on the final result, but initialization is necessary when writing programs, so we report this situation to the tool developers.

Some *If conditions* are regarded as deadcode by the Type 2 tool, but we find it is attached during compilation and it can have impact on final results

```
if (useStartAsPosition) atPos(start)(repeatedApplication(t))
                    else atPos(t.pos.start, t.pos.point)(repeatedApplication(t))
```

At the same time, some if condition dead codes are not real dead codes, because the else conditions are not consider as deadcode by type 2 tool. We give feedback to the tool developers to improve their tools.

We also find some deadcode in *Case* situation. Some of them can be treated the same as if condition. Also, some of them may not have impact on the final result but we cannot remove it. For instance,

```
case rt: RefinedType if !(rt =:= highType) && !(checkedCombinations contains
rt.parents) =>
                          // might mask some inconsistencies -- check overrides
                          checkedCombinations += rt.parents
                          val tsym = rt.typeSymbol
                          if (tsym.pos == NoPosition) tsym setPos member.pos
                          checkAllOverrides(tsym, typesOnly = true)
                      case _ =>
```

The last case match the other conditions and do nothing during the execution, but it is useful. We report this situation to the Type 2 tool developers.

For type 3 deadcodes, we find there’s not much io operation during compiling procedure and we do not find any type 3 deadcodes this time.

3.6 The Result List of Three Types of Deadcode

Our submission of deadcode to Typesafe is to remind them there are deadcodes in their Scala compiler. During our testing and verification, we submitted the following dead codes to Typesafe Engineers.

Potential Type 1 Deadcode

We submit the following potential deadcodes on December 21st 2015, because they are with no caller. The list table gives the paths of files with all codes dead. The codes in setter.scala, ClosureElimination.scala and InlineErasure.scala are all dead codes which have never been touched during compilation process by using our test programs and have never been called by other part of Scala source codes.

Potential Deadcode
src\library\scala\annotation\meta\setter.scala
src\compiler\scala\tools\nsc\backend\opt\ClosureElimination.scala

src\compiler\scala\tools\nsc\transform\InlineErasure.scala

The Engineer Jason Zaugg reply our analysis. The file setter.scala defines an annotation and contains no code at all. The logic in ClosureElimination.scala is only executed with -optimize (or the more fine-grained option, -Yclosure-elim) compiler flag. We would need to run our code coverage analysis with a variety of compiler options, together with a variety of Scala programs, to avoid the false positive here.

InlineErasure.scala doesn't define any code either, but it is indeed dead code, the trait is never used. It was added in the early development of Value Classes as a placeholder, but was never fleshed out. It is removed in the latest 2.12.x branch as part of removal of unused code in the compiler and standard library.

We then submit the following potential deadcode to Typesafe on Apr 17, 2016 and given our analysis results to them. The list table gives files with all codes dead.

Potential Deadcode	***Analysis Results***
src\compiler\scala\tools\nsc\backend \Platform.scala	It is completely base class definition.
src\compiler\scala\tools\nsc\transform \SampleTransform.scala	It is never called by other part in Scala compiler
src\compiler\scala\tools\reflect \FormatInerpolator.scala	It is never called by other part in Scala compiler
src\compiler\scala\tools\nsc\util \WorkScheduler.scala	It is never called by other part in Scala compiler
src\compiler\scala\tools\nsc\util \Interruptreq.scala	It only have some class call
src\compiler\scala\tools\nsc\util \Spec.scala	It is cmd relevant tool but never called during execution.

One of their Engineer Adriaan reply us and says that our analysis are all correct. Although, these base classes or tool classes are required parts of the compiler API, implementation or tool chain so they won't move this point, this remind them to pay attention to these codes during the developing procedure.

We now give the following Type 1 deadcode. All codes that are useless during the compiling process are considered dead code.

Code with no caller or within a trace with no caller
src\compiler\scala\tools\nsc\NewLinePrintWriter.scala
src\compiler\scala\tools\nsc\backend\WorklistAlgorithm.scala
src\compiler\scala\tools\nsc\transform\InlineErasure.scala
src\compiler\scala\reflect\quasiquotes\Placeholders.scala
src\compiler\scala\tools\nsc\transform\SampleTransform.scala
src\compiler\scala\tools\reflect\FormatInterpolator.scala
src\compiler\scala\tools\nsc\MainTokenMetric.scala
src\compiler\scala\tools\reflect\StdTags.scala

Code whose call trace has been deleted
src\compiler\scala\tools\nsc\backend\icode\analysis\CopyPropagation.scala

src\compiler\scala\tools\nsc\backend\icode\analysis\TypeFlowAnalysis.scala
src\compiler\scala\tools\nsc\backend\icode\Linearizers.scala

Code executed outside compiling process
src\compiler\scala\tools\nsc\ClassPathMemoryConsumptionTester.scala
src\compiler\scala\tools\nsc\classpath\ZipAndJarFileLookupFactory.scala
src\compiler\scala\tools\nsc\util\WorkScheduler.scala
src\compiler\scala\tools\nsc\util\InterruptReq.scala
src\compiler\scala\tools\cmd\*
src\compiler\scala\tools\ant\*
'' Stands for all files under the directory*

Code having only structural function
src\compiler\scala\tools\ant\sabbus\CompilationFailure.scala
src\compiler\scala\reflect\reify\utils\Utils.scala
src\compiler\scala\tools\nsc\backend\Platform.scala
src\compiler\scala\tools\nsc\backend\icode\Printers.scala
src\compiler\scala\tools\nsc\backend\icode\TypeKinds.scala
src\compiler\scala\tools\nsc\backend\jvm\analysis\package.scala
src\compiler\scala\tools\nsc\classpath\DirectoryFlatClassPath.scala

Potential Type 2 Deadcode

When we analyze type 2 deadcode, we find the following potential deadcode.

/compiler/scala/tools/nsc/typechecker/MethodSynthesis.scala, line-423
/compiler/scala/tools/nsc/transform/patmat/MatchOptimization.scala, line-131
/compiler/scala/tools/nsc/backend/jvm/opt/InlineInfoAttribute.scala, line-41 to line-86
/compiler/scala/tools/nsc/backend/jvm/BCodeHelpers.scala, line-68 to line-75

4. Conclusion

Through our experiments, we can conclude that compiler plugin is feasible avenue for source-level dynamic dead code detection. Type 1 dead code has been successfully detected in the scalac source. We use Scala IDE to check whether the function is called during the compilation process or it is isolated. We use keyword analysis to judge the potential dead codes.

Then, dealing with Type 2 and 3, the icode method is useful for writing the tool. We check the call trace and check if it is real deadcode. Our call trace method is useful for three types of dead codes.

Although we use different method to check the different types of dead codes, there still remains some codes which are hard to make judgements. The reasons for this is `as follows: 1) Some of codes may have extra uses by developer during compilation, such as helping them debug the codes, using them to help developer upgrade the scala version. 2) Some of codes have if condition and case situation. We cannot make sure these statements indeed never happen in more complex situations. 3) Some of codes may related to library operation.

For the time-consuming problem of Scala compiling procedure, we think it’s not because of the large amount of deadcodes in Scala compiler. There too many phases during compilation. When we compile program, this disadvantage makes the compiling procedure costs a lot of time.