Dead Code Detection in Scala Compiler Source

Yi Liu, Yu Shi, Changjiang Zhong, Kenny Q. Zhu
ADAPT Lab, Department of Computer Science & Engineering
Shanghai Jiao Tong University

Abstract

Scala is a Java-like programming language which unifies object-oriented and functional programming. However, compilation process of a Scala program is mush slower than a similar java program. We want to detect whether there is dead code in Scala compiler source for some certain set of programs as dead code may consume lots of compilation time. Our Scala compiler plugin is the core of our detection work, which outputs positions of executed code in Scala compiler source. These traces eventually help us seek out the dead code in Scala compiler source. Our evaluation results verify based on the set of testing programs we random pick, there is pretty much dead code.

1 Introduction

Compilation process of Scala programs is much slower than java, due to more than 20 compilation phases and a large number of type inferences, type checkings, transformations on AST. The central data structure of the compiler is class scala.tools.nsc.ast.Tree. Trees are immutable structures, however, they are decorated with symbol and type information in a mutable way. We wish to find dead code in these related class for better performance.

We suspect there are three types of dead code in the Scala compiler source: obviously, first kind of dead code we could think of is code that not executed (we call *type 1 dead code*). Second kind of dead code that we call *type 2 dead code* is executed assignment code but does nothing to results. And the last kind of dead code is I/O relevant code in type 2 dead code, which we call *type 3 dead code*.

There are about 162,000 lines of code in Scala compiler source, and existing off-the-shelf tools don't work very well for our purpose (we tried to use *Linter* to detect dead code static, but the code detected is little and it's hard to detect type 2 and type 3 dead code). So by Scala compiler plugin, we create a dynamic detection tool to track the compilation process. We make that every executed line in Scala compiler source is denoted by the file name and the line number when we use Scala compiler to compile some program. Thus we can compare this result to the whole Scala compiler source and find out dead code.

In summary, we make two main contributions.

i. We write a Scala compiler plugin to transform the AST of compiled programs after "typer" phase during compilation process, adding a piece of "println" code in front of every piece of executable code in compiled programs.

ii. We write a Scala program to list all dead code in Scala compiler source, using the traces generated by our plugin.

2 Approaches

2.1 Previous attempts

We first use a static detection tool to detect the dead code. However, most of the detected code have little impact on the compilation speed since it doesn't show up in the most frequently used classes. And it's hard to detect type 2 and 3 dead code from static analysis after trying. Dynamic method is a better way for doing so.

2.2 Compiler plugin technology

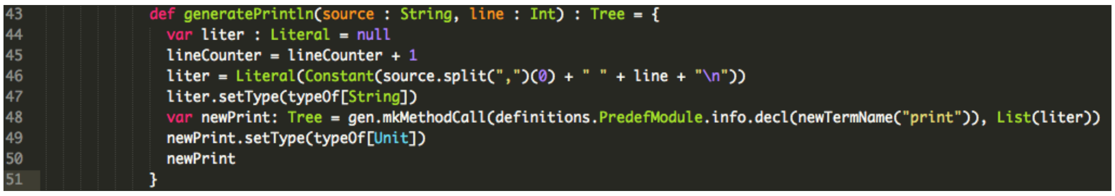
Compiler plugin lets you modify the behavior of the compiler itself without needing to change the main Scala distribution. http://www.scala-lang.org/old/node/140 refers to a web page where you can get some introduction to Scala compiler plugin.

Basic idea of our plugin is to transform the AST of Scala compiler source by adding "println" expression in front of every executable line. Arguments of the "println" expression are file names and line numbers of these lines. For we want to track the behavior of Scala compiler, we use *ant* to build Scala compiler together with our plugin. The compiler built by our way not only has its original function but also can output the locations of every executed line.

Our plugin starts from the whole AST of a source file, each node of which denotes a construct occurring in the source code. We encounter three main problems :

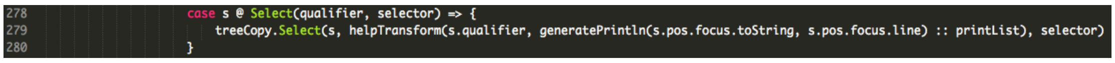
1. **How can we create the "println" expression and add it into the original AST?**
2. **How can we add "println" expression as complete as possible**
3. **How can we avoid adding expression at wrong places?**

Problem 1 is not complex. Figure 2.1 and 2.2 show how we generate a "println" expression and add it in front of some piece of code.



```scala
def generatePrintln(source : String, line : Int) : Tree = {
  var liter : Literal = null
  lineCounter = lineCounter + 1
  liter = Literal(Constant(source.split(",")(0) + " " + line + "\n"))
  liter.setType(typeOf[String])
  var newPrint: Tree = gen.mkMethodCall(definitions.PredefModule.info.decl(newTermName("print")), List(liter))
  newPrint.setType(typeOf[Unit])
  newPrint
}
```

Figure 2.1 Code for generating a "println" expression

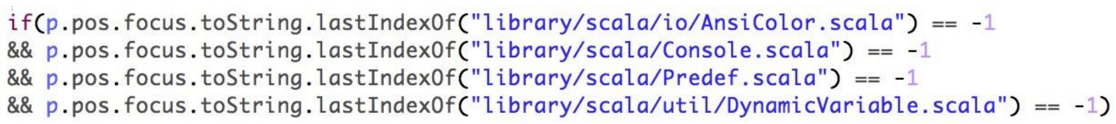


```scala
case s @ Select(qualifier, selector) => {
  treeCopy.Select(s, helpTransform(s.qualifier, generatePrintln(s.pos.focus.toString, s.pos.focus.line) :: printList), selector)
}
```

Figure 2.2 Code for transform a "Select" type AST

Our solution to problem 2 is gained from lots of trial. In the beginning, we used our plugin on a toy hello program to test whether we could add "println" expressions successfully. Then we applied our plugin to Scala compiler source. Of course, original results were pretty disappointing because we didn't anticipate a lot of cases which usually resulted in that we only added expressions in front of a block but failed to add them before code in the block. So we had to revise our plugin source code to deal with as many cases as possible. Though our plugin is not perfect enough to add "println" expressions in front every piece of executable code, results show that it is powerful enough to help us find out the dead code.

Problem 3 may be confusing. For example, what if we add the "println" expression in the function which implements "println"? Answer is obvious : endless recursion. So, not all of Scala source is applied to the plugin. We make our plugin to exclude some files. Figure 2.3 shows our plugin's principle of exclusion.



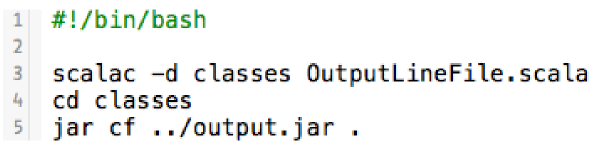
```
if(p.pos.focus.toString.lastIndexOf("library/scala/io/AnsiColor.scala") == -1
&& p.pos.focus.toString.lastIndexOf("library/scala/Console.scala") == -1
&& p.pos.focus.toString.lastIndexOf("library/scala/Predef.scala") == -1
&& p.pos.focus.toString.lastIndexOf("library/scala/util/DynamicVariable.scala") == -1)
```

Figure 2.3 Plugin excludes some files from being applied

2.3 Build our own Scala compiler

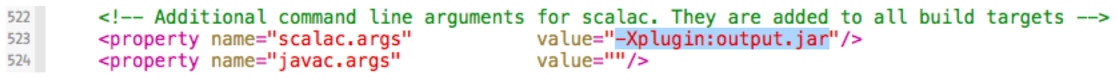
Scala source we use is downloaded from https://github.com/scala/scala. We use *ant* to build our Scala compiler. Figure 2.5 shows how to tell *ant* to build Scala compiler with plugin we write. *output.jar* is a jar file of the file formed by the compilation result of our plugin. Figure 2.4 is a shell script about how to generate the jar file. Concrete procedure is :

i. Generate *output.jar* as Figure 2.4 shows
ii. Move the jar file to root directory of Scala source
iii. Add argument showed in Figure 2.5 into file build.xml which is in the root directory of Scala source
iv. Keep your terminal in the root directory of Scala source, run command "*ant*", your can see script of our Scala compiler is at *scala/build/quick/bin*.



```
#!/bin/bash

scalac -d classes OutputLineFile.scala
cd classes
jar cf ../output.jar .
```

Figure 2.4 Generation of *output.jar*



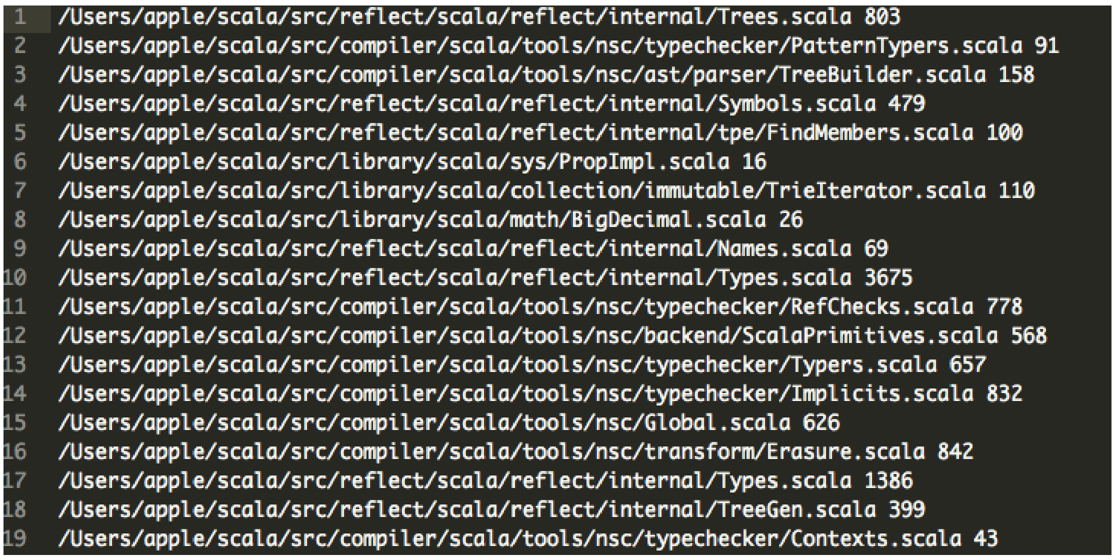
```
<!-- Additional command line arguments for scalac. They are added to all build targets -->
<property name="scalac.args"          value="-Xplugin:output.jar"/>
<property name="javac.args"           value=""/>
```

Figure 2.5 How to build Scala compiler with plugin

2.4 Collect the execution traces and find out the dead code

Now, we have our own Scala compiler. In terminal, we use it to compile our testing programs, and redirect the output into a text document. Figure 2.6 shows a part of one text document. **Code in source that compiler use to compile testing programs is denoted by file name and line number.** Every line of the documents is called a piece of trace. The size of these documents are large, for a lot of traces will appear tens of thousands of times. So clearing is necessary. In general, if we reserve every trace only once, we can simplify the document to about 2MB. Combine all traces generated by our testing programs and simplify it into one text document. Then run our tool to find out the dead code. Concrete procedure is :

i. Put all trace documents in one file
ii. Use *deleteRedun.java* to get a merged trace document
iii. Use *tool.scala* to get final results



```
/Users/apple/scala/src/reflect/scala/reflect/internal/Trees.scala 803
/Users/apple/scala/src/compiler/scala/tools/nsc/typechecker/PatternTypers.scala 91
/Users/apple/scala/src/compiler/scala/tools/nsc/ast/parser/TreeBuilder.scala 158
/Users/apple/scala/src/reflect/scala/reflect/internal/Symbols.scala 479
/Users/apple/scala/src/reflect/scala/reflect/internal/tpe/FindMembers.scala 100
/Users/apple/scala/src/library/scala/sys/PropImpl.scala 16
/Users/apple/scala/src/library/scala/collection/immutable/TrieIterator.scala 110
/Users/apple/scala/src/library/scala/math/BigDecimal.scala 26
/Users/apple/scala/src/reflect/scala/reflect/internal/Names.scala 69
/Users/apple/scala/src/reflect/scala/reflect/internal/Types.scala 3675
/Users/apple/scala/src/compiler/scala/tools/nsc/typechecker/RefChecks.scala 778
/Users/apple/scala/src/compiler/scala/tools/nsc/backend/ScalaPrimitives.scala 568
/Users/apple/scala/src/compiler/scala/tools/nsc/typechecker/Typers.scala 657
/Users/apple/scala/src/compiler/scala/tools/nsc/typechecker/Implicits.scala 832
/Users/apple/scala/src/compiler/scala/tools/nsc/Global.scala 626
/Users/apple/scala/src/compiler/scala/tools/nsc/transform/Erasure.scala 842
/Users/apple/scala/src/reflect/scala/reflect/internal/Types.scala 1386
/Users/apple/scala/src/reflect/scala/reflect/internal/TreeGen.scala 399
/Users/apple/scala/src/compiler/scala/tools/nsc/typechecker/Contexts.scala 43
```

Figure 2.6 Output of compilation of testing programs

3 Experiments and analysis

3.1 Data sets

We pick seven projects and many individual Scala programs to test our program. Detailed information about them is listed in Table 1.

Table 1. Testing data

project name	introduction	number of code lines
scalastyle	Scalastyle examines your Scala code and indicates potential problems with it.	8429
scala-dddbase	scala-dddbase-develop is a library used to develop application based on Domain Driven Design advocated by Eric Evans.	2732
scala-stm	scala-stm is a lightweight software transactional memory for Scala, inspired by the STMs in Haskell and Clojure.	12458
Scala-Algorithms	Java to Scala translations of around 50 algorithms from Robert Sedgewick and Kevin Wayne's website for their book Algorithms.	4766
scala-chart	scala-chart is a scala library for creating and working with charts.	2957
scala-cassandra	Implementation of a Scala wrapper over the DataStax Java Driver for Cassandra	308
scala-parser-combinators	Scala Standard Parser Combinator Library	1880
170 individual programs	Individual scala programs, mostly from scala reference books' code example, covering most basic syntaxes.	4943

3.2 Type 1 Experiments and analysis

We use the scalac we build to compile testing programs, and get a text document for every program (or project) which contains execution traces. The text documents are quite large, scaling from 50GB to 1.5TB. There are many traces appearing tens of thousands of times. After deleting the repeated traces, we get a about 2MB document for every testing program (or project). Put all trace documents together, and delete the repeated traces. The executed lines in source files for our testing data set are eventually collected. The last work is to get the complements of these traces. Our tool solves this problem. For detailed use, please check README in ...

Some significant results from our testing data are listed below. **Note that blank lines and lines of comments in source files are not counted.**

All sources of Scala programming language are in the file “scala/src”. Necessary code for compilation process are in “compiler”, “reflect” and “compiler”. So the rest of code, which are for IDE configuration, doc generation and so on, are regarded as dead code for we didn’t test these parts. Table 2 lists the the dead code distribution in files in “scala/src”.

Table 2. Dead code distribution in “scala/src”

file path	number of total lines	number of total dead lines	percentage of dead lines
scala/src/actors	2908	2908	1.000
scala/src/build	568	568	1.000
scala/src/compiler	72206	32704	0.536
scala/src/eclipse	0	0	0.000
scala/src/ensime	0	0	0.000
scala/src/forkjoin	0	0	0.000
scala/src/intellij	0	0	0.000
scala/src/interactive	2941	2941	1.000
scala/src/library	35750	26237	0.763
scala/src/library-aux	36	36	1.000
scala/src/manual	1460	1460	1.000
scala/src/partest-extras	611	611	1.000
scala/src/partest-javaagent	0	0	0.000
scala/src/reflect	31039	12340	0.460
scala/src/repl	5193	5193	1.000
scala/src/repl-jline	241	241	1.000
scala/src/scaladoc	7471	7471	1.000
scala/src/scalap	2306	2306	1.000

As compilation process of Scala programs only relates to code in “compiler”, “library” and “reflect” directory, our following analysis will focus on the three files. Table 3 and Table 4 show files that contains most dead code.

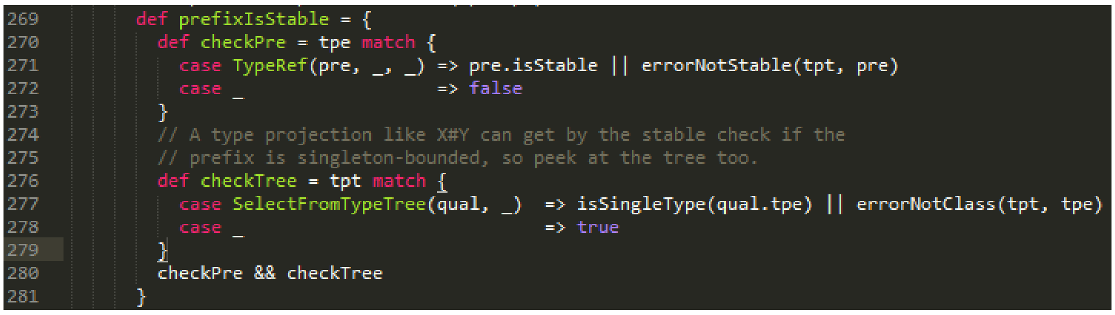
Table 3. Ten files with most dead lines

file path	number of total lines	number of total dead lines	percentage of dead lines
scala/src/reflect/scala/reflect/runtime/JavaMirrors.scala	1034	1034	1.000
scala/src/library/scala/collection/parallel/ParIterableLike.scala	967	967	1.000
scala/src/compiler/scala/tools/nsc/symtab/classfile/ICodeReader.scala	959	959	1.000
scala/src/compiler/scala/tools/nsc/typechecker/Typers.scala	4574	917	0.354
scala/src/reflect/scala/reflect/internal/Printers.scala	1143	822	0.747
scala/src/compiler/scala/tools/nsc/backend/jvm/BCodeBodyBuilder.scala	1020	708	0.772
scala/src/compiler/scala/tools/nsc/javac/JavaScanners.scala	757	623	0.823
scala/src/compiler/scala/tools/nsc/typechecker/ContextErrors.scala	1067	607	0.569
scala/src/library/scala/collection/immutable/Vector.scala	1028	586	0.694
scala/src/reflect/scala/reflect/internal/Types.scala	3172	572	0.285

Table 4. Five files with most dead code in three directories

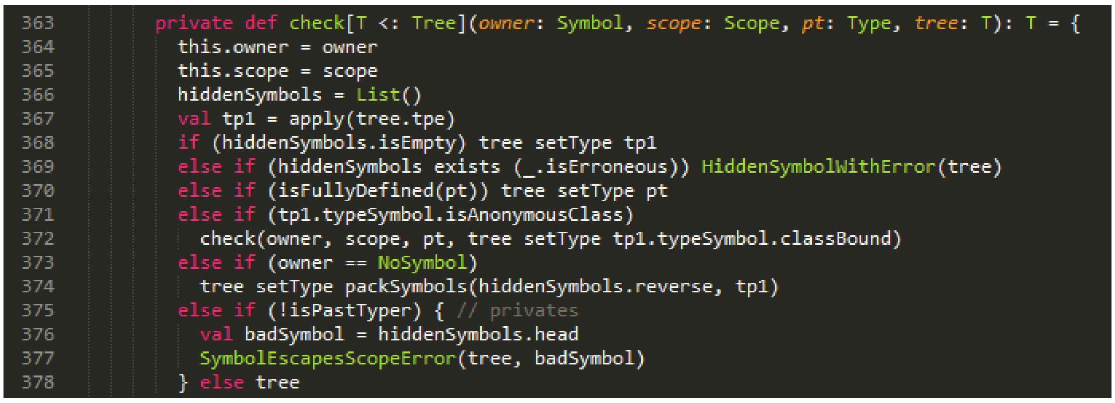
	file path	number of total lines	number of total dead lines	percentage of dead lines
src/compiler	compiler/scala/tools/nsc/symtab/classfile/ICodeReader.scala	959	959	1.000
	compiler/scala/tools/nsc/typechecker/Typers.scala	4574	917	0.200
	compiler/scala/tools/nsc/backend/jvm/BCodeBodyBuilder.scala	1020	708	0.694
	compiler/scala/tools/nsc/javac/JavaScanners.scala	757	623	0.823
	compiler/scala/tools/nsc/typechecker/ContextErrors.scala	1067	607	0.569
src/library	library/scala/collection/parallel/ParIterableLike.scala	967	967	1.000
	library/scala/collection/immutable/Vector.scala	1028	586	0.570
	library/scala/collection/concurrent/TrieMap.scala	872	582	0.667
	library/scala/collection/parallel/mutable/ParArray.scala	575	575	1.000
	library/scala/collection/parallel/RemainsIterator.scala	523	523	1.000
src/reflect	reflect/scala/reflect/runtime/JavaMirrors.scala	1034	1034	1.000
	reflect/scala/reflect/internal/Printers.scala	1143	822	0.719
	reflect/scala/reflect/internal/ReificationSupport.scala	966	707	0.732
	reflect/scala/reflect/internal/Types.scala	3172	572	0.180
	reflect/scala/reflect/api/Trees.scala	812	491	0.605

Figure 1-3 are source code fragments that contain dead code. In figure 1, our tool tells 272,276,277,278 are dead code. In figure 2, 369-374, 376, 377 are dead code. In figure 3, 48, 52, 54 are dead code. As you can see, our tool is not perfect enough for there are some omissions, including a right brace and a condition statement. But the dead code we find is very likely to be dead code.



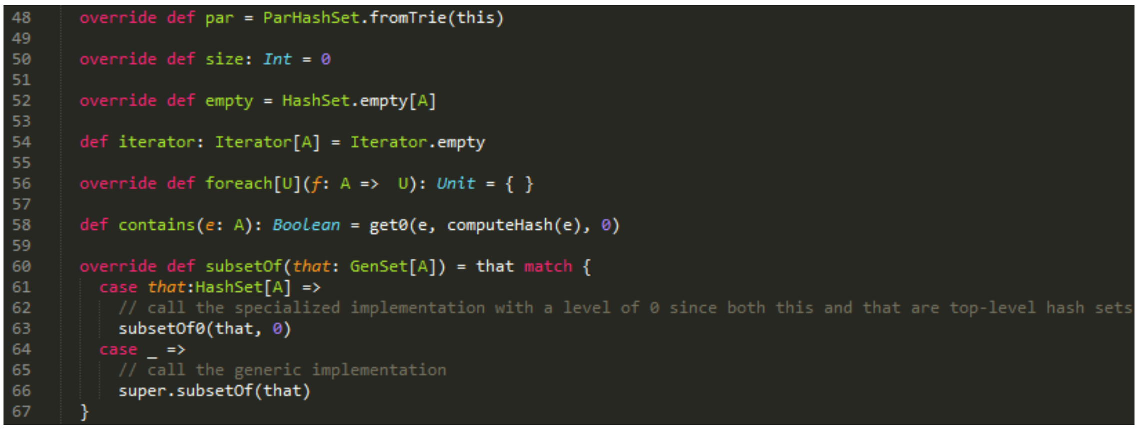
```
269    def prefixIsStable = {
270      def checkPre = tpe match {
271        case TypeRef(pre, _, _) => pre.isStable || errorNotStable(tpt, pre)
272        case _                  => false
273      }
274      // A type projection like X#Y can get by the stable check if the
275      // prefix is singleton-bounded, so peek at the tree too.
276      def checkTree = tpt match {
277        case SelectFromTypeTree(qual, _)  => isSingleType(qual.tpe) || errorNotClass(tpt, tpe)
278        case _                            => true
279      }
280      checkPre && checkTree
281    }
```

Figure 1. Dead code segment in Typers.scala



```scala
private def check[T <: Tree](owner: Symbol, scope: Scope, pt: Type, tree: T): T = {
  this.owner = owner
  this.scope = scope
  hiddenSymbols = List()
  val tp1 = apply(tree.tpe)
  if (hiddenSymbols.isEmpty) tree setType tp1
  else if (hiddenSymbols exists (_.isErroneous)) HiddenSymbolWithError(tree)
  else if (isFullyDefined(pt)) tree setType pt
  else if (tp1.typeSymbol.isAnonymousClass)
    check(owner, scope, pt, tree setType tp1.typeSymbol.classBound)
  else if (owner == NoSymbol)
    tree setType packSymbols(hiddenSymbols.reverse, tp1)
  else if (!isPastTyper) { // privates
    val badSymbol = hiddenSymbols.head
    SymbolEscapesScopeError(tree, badSymbol)
  } else tree
```

Figure 2. Dead code segment in Typer.scala



```scala
override def par = ParHashSet.fromTrie(this)

override def size: Int = 0

override def empty = HashSet.empty[A]

def iterator: Iterator[A] = Iterator.empty

override def foreach[U](f: A =>  U): Unit = { }

def contains(e: A): Boolean = get0(e, computeHash(e), 0)

override def subsetOf(that: GenSet[A]) = that match {
  case that:HashSet[A] =>
    // call the specialized implementation with a level of 0 since both this and that are top-level hash sets
    subsetOf0(that, 0)
  case _ =>
    // call the generic implementation
    super.subsetOf(that)
}
```

Figure 3. Dead code segment in HashSet.scala

We assume that all code in source files are dead if there is no testing program. Then we count the dead code we find by our tool with the testing data getting larger, and the result is showed in the following figure. A few testing code need about half of code in source files. Once testing data gets a bit large, the dead code we can find is quite little. One possible reason is that there is actually not so many dead code after testing data becomes that large, but our plugin is not powerful enough to output line number of all executed lines.

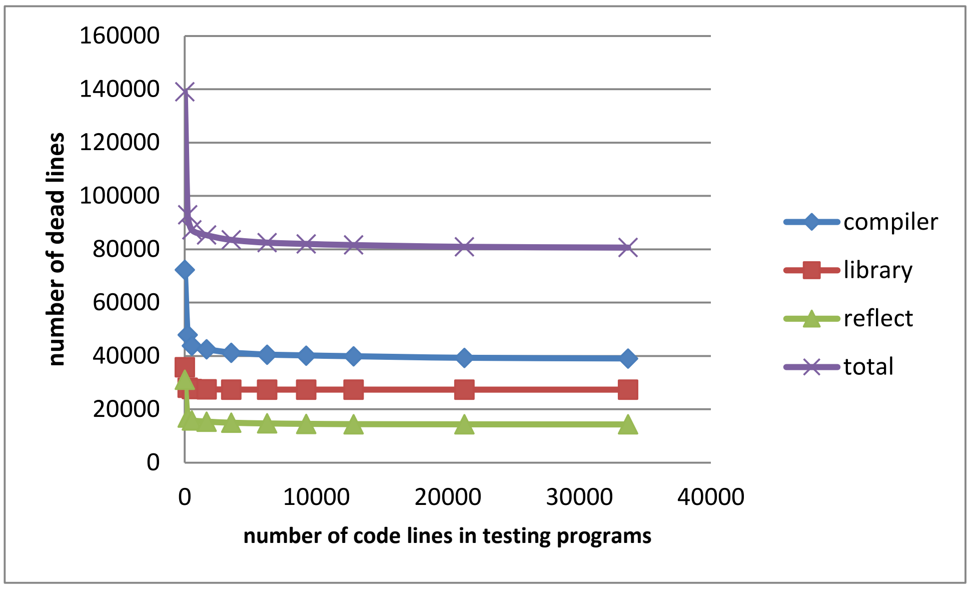


Figure 4. Dead code detected by different number of testing data

For Type 2 dead code, we still use scala compiler plugin to transform the compiler source and print out dynamic data flow at run time. In Type 1, we print the source file name and line number of executed code. But in Type 2, we print read-and-write, or data dependencies that happen in each line. Our compiler, built with the plugin, produces a file of data flow during compilation. By analyzing this file, we find out useless data dependencies, which doesn't contribute to the final result of compilation.

3.3 Type 3 Experiments and analysis

3.3.1 Data flow representation

A data flow in our Type 2 detection represents one data dependency in the program. Each data flow occupies one line in the data flow file.

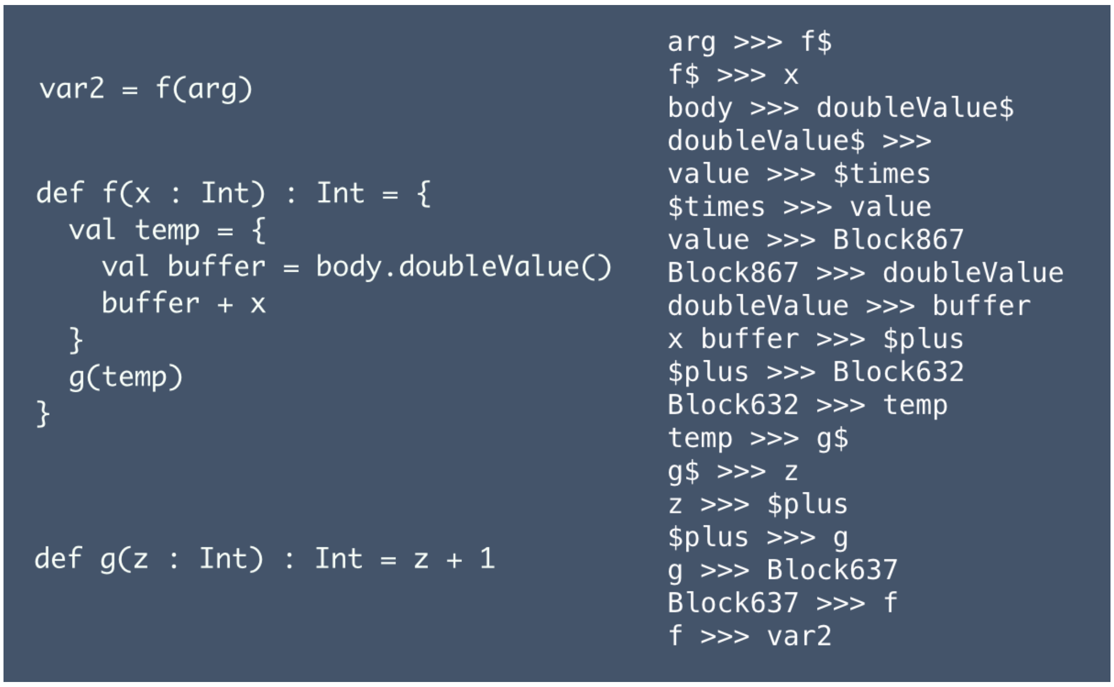
Each data flow consist of three parts:

Left hand side: Names of AST nodes which determines the value of the AST node in the right hand side.

Right hand side: Name of AST node whose value depends on the left hand side AST nodes.

Arrow: A notation ">>>" which indicates the data flow direction.

Here is an example, showing data flow of a small program.



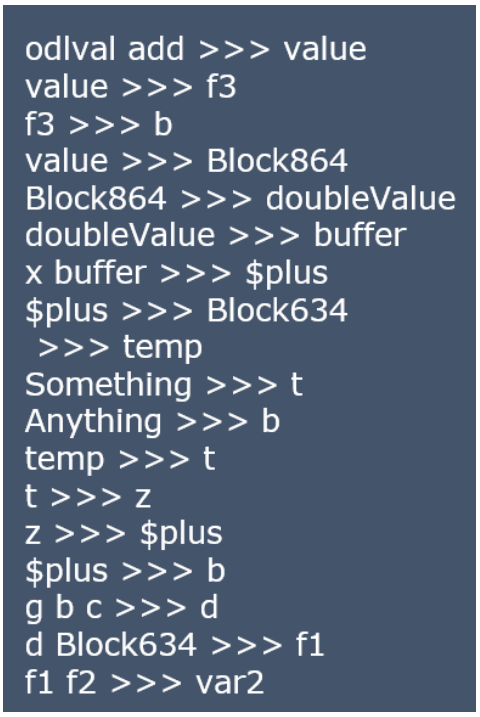
```
var2 = f(arg)


def f(x : Int) : Int = {
  val temp = {
    val buffer = body.doubleValue()
    buffer + x
  }
  g(temp)
}



def g(z : Int) : Int = z + 1
```

```
arg >>> f$
f$ >>> x
body >>> doubleValue$
doubleValue$ >>>
value >>> $times
$times >>> value
value >>> Block867
Block867 >>> doubleValue
doubleValue >>> buffer
x buffer >>> $plus
$plus >>> Block632
Block632 >>> temp
temp >>> g$
g$ >>> z
z >>> $plus
$plus >>> g
g >>> Block637
Block637 >>> f
f >>> var2
```

Figure 5.Sample data flow of a simple program

To represent data flow with AST nodes, we first must give the AST nodes names. In scala compiler, some ASTs have symbols, which can be used as their names. Such ASTs including Ident, Select, DefDef, ValDef, etc. Others don't have their own symbols, like Block, Apply, Return, If, Match, New, Try, etc. For AST with a symbol, we just use its symbol as its name in data flow. For those without symbol, we use their type name, appended by their node id in the source file as their name.

3.3.2 Data flow search



```
odlval add >>> value
value >>> f3
f3 >>> b
value >>> Block864
Block864 >>> doubleValue
doubleValue >>> buffer
x buffer >>> $plus
$plus >>> Block634
 >>> temp
Something >>> t
Anything >>> b
temp >>> t
t >>> z
z >>> $plus
$plus >>> b
g b c >>> d
d Block634 >>> f1
f1 f2 >>> var2
```

Figure 6.Sample data flows for search

To search the data flow, first we need a target node, which is the final result of the compiled program. Then we start the following sequence:

1. Find the line which has the target node as its right hand side.

2. Put the target node into the required list.

3. Go one line upwards in the data flow, check if the right hand side node is in required list. If so, take it out from the required list and replace it with nodes on the left hand side, mark this line as useful. Otherwise, do nothing.

4. Repeat step 3 until the whole data flow file is processed.

In the end, those lines not marked as useful are redundant data flow. Find out the corresponding lines in compiler source, we get Type 2 dead code. To do this, we can just append the position in compiler source for each line of data flow, for example:

f >>> var2   scala/reflect/tools/nsc/Typers.scala 125

Which means data flow, *f >>> var*, happens in line 125 of source file *scala/reflect/tools/nsc/Typers.scala*. This helps us find the corresponding position in compiler source easily. For simplicity, the position information is omitted in examples here.

To make our sample data flow more visible, we can draw a tree from the sequence. The arrows in the tree shows directions, or data dependencies. Latest dependencies are represented in full line, and previous dependencies are in dash lines. In our example, variable *b* is assigned three times. However, only the latest one, *$plus >>> b* is used by a later data flow *g b c >>> d*. Other two previous assignments, *Anything >>> b* and *f3 >>> b* are redundant, because their effects are overlapped by *$plus >>> b*. Regard dash lines as disconnection and full lines as connection, then all data dependencies contributing to the final result *var2* are included in the connected tree which has *var2* as its root node.

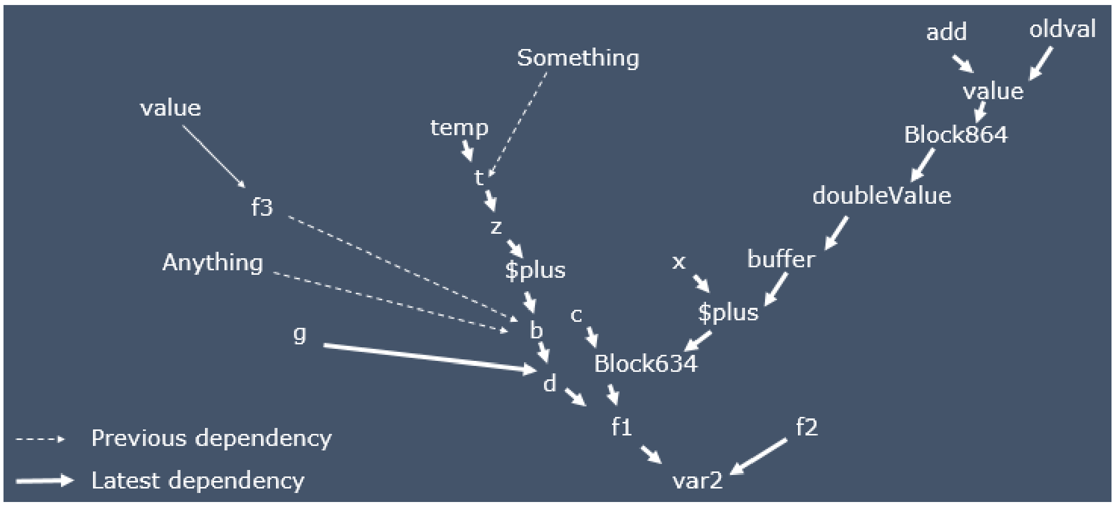


Figure 7.Sample data flow of a simple program

3.3.3 Order of data flow

To make the search easy, efficient and precise, we want to search the data flows only in one direction, from down to top. This requires a careful arrangement of positions to insert the "*println*" statement.

Here's an example of where to put the "*println*" in the source code. Suppose we have a function *f*, whose return type is *Int*, and it calls another function g at the tail position as its return value. Give the right hand side block of *f* a name, *Block637*. Because *g(temp)* is the last line in this block, or the expression of this block, so the the value of *Block637* equals to the value of *g(temp)*. So we get two data flows, *g >>> Block637* and *Block637 >>> f*. Also, we have another data flow in the body of function *g*, telling what return value of *g* depends on, and it should be in this form:

something >>> g. Because we want to search the data flow from down to top all along the way, these three data flows should be printed out in this way:

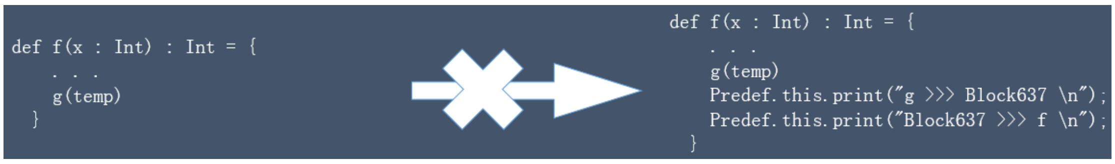
something >>> g

g >>> Block637

Block637 >>> f

Note that *something >>> g* is inserted in the body of definition of *g*. But the other two are inserted in the definition of *f*.

To make *something >>> g* above the other two, it is a straightforward idea to put the "*println*"s as follows, but this will cause an error because function *f* will use "*println*" as its return value.

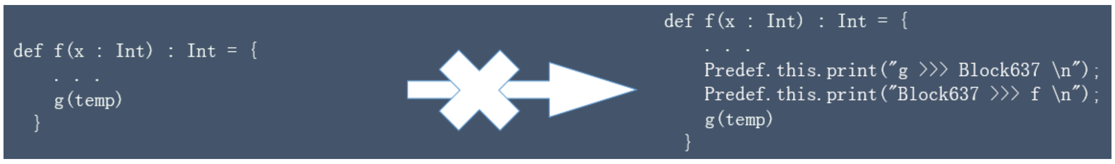


```
def f(x : Int) : Int = {
    . . .
    g(temp)
  }
```

```
def f(x : Int) : Int = {
    . . .
    g(temp)
    Predef.this.print("g >>> Block637 \n");
    Predef.this.print("Block637 >>> f \n");
  }
```

Figure 8.First way to insert "*println*"s (Wrong)

Another choice is to insert "*println*"s before function call of g.



```
def f(x : Int) : Int = {
    . . .
    g(temp)
  }
```

```
def f(x : Int) : Int = {
    . . .
    Predef.this.print("g >>> Block637 \n");
    Predef.this.print("Block637 >>> f \n");
    g(temp)
  }
```

Figure 9.Second way to insert "*println*"s (Wrong)

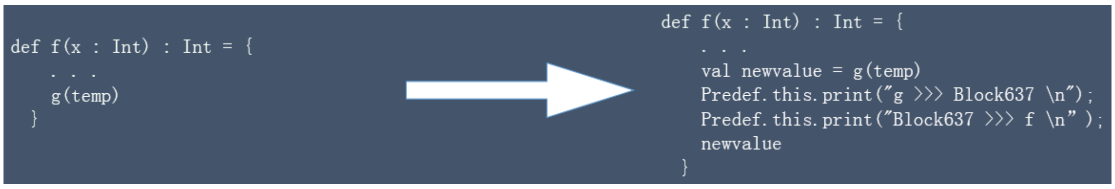
But this will result in a wrong order:

g >>> Block637

Block637 >>> f

something >>> g

To solve this problem, we first store the value of *g(temp)* into an value "*newvalue*" when it is called. Then insert *Block637 >>> f* and *something >>> g*. Finally, we use "*newvalue*" as the return value of f. In this way, the order of data flow is maintained and f also has a correct return value.



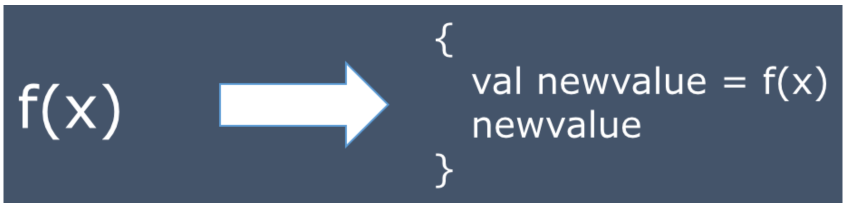
```
def f(x : Int) : Int = {
    . . .
    g(temp)
  }
```

```
def f(x : Int) : Int = {
    . . .
    val newvalue = g(temp)
    Predef.this.print("g >>> Block637 \n");
    Predef.this.print("Block637 >>> f \n" );
    newvalue
  }
```

Figure 10.The right way to insert "*println*"s

3.3.4 Challenges

Maintaining the order of data flow is our difficulty in implementing Type 2 detections. As is shown in 3.3.4, to produce a correct data flow, we need transformations like this:



```
f(x)
```

```
{
   val newvalue = f(x)
   newvalue
}
```

Figure 11.Transformation frequently needed in Type2 detection

Choosing a right phase to do this transformation is important. If we do this before “typer” phase, we cannot get the right type tree when creating “newvalue” definition in the AST. However, if we do this after “typer” phase, we have to deal with symbols and types by ourselves, which is very complicated and can cause various errors easily.

Also, in the compiler source, many functions are tail-recursion. Maintaining the order for data flow also requires transformation at the tail call. But after the transformation, the function is no longer tail-recursion. This disable the compiler to do tail-recursion optimization when compiling the source, which results in stack overflow error at runtime.

3.3.5 Preliminary Type 2 Results

Because of the problems above, we cannot get data flow as expected. The data flow are out of order and broken. So we can only get some preliminary results.

Table 5.Preliminary Type 2 Results

File Name	Useful data dependencies
Driver.scala	10/49
ConsoleReporter.scala	3/11
Typers.scala	49/13065

3.4 Type 3 Experiments and analysis

The detection of Type 3 dead code should be based on Type 2. To find out I/O operations during compiling, we 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, we should look for basic I/O function names defined in java, such as “java.io.FileWriter”, “write”, etc.

Because of existing problems in Type 2, we cannot get result of Type 3 dead code, but once those problems are solved, Type 3 will be an easy work.

1. 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.

However, we still have caveats with Type 2 and 3. First, maintaining order of execution traces is hard, given tail-recursion and other complex features in Scala. Second, transforming original AST may cause symbol mismatches. Thus data flow currently produced can be short and broken.