Dead Code Detection of Real World Scala Applications-PhaseII

Advisor: Prof. Kenny Zhu
Team Members: Yu Shi, Yi Liu, Pei Lv, Benxuan, Zhang

October 2, 2016

Contents

1 Introduction

Scala is a programming language uniting both functional and objective-oriented styles, which gains popularity due gradually these days. To guarantee the efficiency of Scala programs, it's better to use an automatic code reviewer or coverage tool. We develop a tool set to do runtime data dependence analysis and detect dead codes in Scala programs.

Compared with existing code coverage tool, our tool can dig into data dependences among variables and values. Scoverage only analyzes usage of source code at runtime, like how many lines of codes are executed during one execution. Our tool does even more. Given a target, which can be either a variable or value in the program, it can find out all codes that contribute to the final value of this target, thus detects useless data flow at run time.

The basic idea of our tool is to change the program with a special compiler or compiler plugin, and make it provide data dependence messages for every operation in the program. Then a small tool will be used to analyze these messages at real time when the program is running.

First we need to consider two essential problems about data dependence analysis

1. How to represent data dependences?
2. How to obtain data dependences at runtime?
3. How to analyze the dependences at runtime?

Each of these questions will be answered respectively for AST and ICode approaches in following sections.

2 Running Example

Through out this report we'll use a simple running example to show how a data dependences can be obtained from the AST or ICode of a program and how to analyze these dependences. The example is a Scala class defines an `add` function, which simply increase the parameter by 1 and return. Also it has another function `callAdd` inside which the `add` function is called and the result is stored into `addResult` value.

```
1  class Add {
2      def add(x: Int): Int = {
3          x + 1
4      }
5
6      def callAdd() {
7          val addResult = add(8)
8      }
9  }
```

3 AST Approach

Abstract Syntax Tree (AST) is an intermediate representation used widely by compilers. Scala provides compiler plugin as a convenient tool for developers to modify ASTs between compiler phases. In this section, we introduce our approach to modify AST so that it can provide data flow information to the runtime analyzing module, as well as the approach for analyzing.

3.1 Represent Data dependences with AST

To represent data dependences, we can make use of information provided by Scala compiler. Since we use the compiler to change the program's behavior, it is very convenient to use informations from AST or ICode. ICode is a intermediate representation in Scala compiler between AST and byte code, which is translated into JVM byte code in the end. This section we focus on representing data dependences using information from ASTs. In section 4.1 we consider how to do it with ICode.

Abstract Syntax Tree (AST) is a tree data structure used by compilers, which representing the parsed structure of a program. First let's see how the function definition `def add(...){...}` is parsed into AST

Now we want to represent the dependences related to value `addResult`. First, notice see that `addResult` is an identifier, corresponding to `Ident` AST in Scala. And the function call `add(8)` is an `Apply` AST in Scala. Using `>>>` to represent dependence, then

```
Apply >>> Ident
```

which means the result of function call `add(8)` determines the value of `addResult`. However, the `Apply` and `Ident` should be given more specific names for analysis. For example, use `addResult` as the name of this `Ident` tree and `add` as the name of this `Apply` tree. Thus the information we provide to the analysis unit should be

```
add >>> addResult
```

Similarly, other tree structures should have their specific names. For tree structures without a inherited suitable name, we will create a synthesis name, for example, a `Block` tree may use the name `Block124`, where `124` is the ID of this `Block` provided by the compiler. Here is the table about how names are given to various ASTs.

In addition, we need another name for each function definition to represent its parameters. We use the function name appended by "$" to represent the parameters of this function. The usage of this special name will be clear in the following example. Here's the full example showing how the value of `addResult` is determined. The "`>>>`" means "determines".

```
8 >>> add$
add$ >>> x
x 1 >>> add
add >>> addResult
```

AST	Name
Apply(args, fun)	Name of fun
DefDef(mods, name, tparams, vparamss, tpt, rhs)	name
Block(stats, expr)	"Block" + AST ID
Return(expr)	"Return" + AST ID
ValDef(mods, name, tpt, rhs)	name
If(cond, thenp, elsep)	"If" + AST ID
Select(qualifier, selector)	selector
Match(selector, cases)	"Match" + AST ID
New(tpt)	"New" + AST ID
Try(block, catches, finalizer)	"Try" + AST ID

The first two lines `8 >>> add$` and `add$ >>> x` say `8` is passed as the argument of function `add`. Here `add$` acts as the bridge that connects `8` and `x`. Note that we don't use `8 >>> x` directly because number `8` is known when the function is called, while name of parameter `x` is only accessible inside the function definition.

3.2 Obtain Data Dependences at Runtime from AST

We've shown how to use information from AST to represent data dependences. Next problem is how can these information be passed to our analyzing module.

The basic idea is, we can change the Scala program, using compiler plugins, to insert some "log" operations inside the program as parts of the program itself. A "log" operation is simply a function call to the analyzing module, taking the data dependence information as a string argument. So that when the program is executed, data dependence information is passed to the analyzing module simultaneously.

Here is how we want to insert the "log" operations into the program.

```
1  class Add {
2      def add(x: Int): Int = {
3          ScalaTrace.logger.log("add$ >>> x")
4          ScalaTrace.logger.log("x 1 >>> add")
5          x + 1
6      }
7
8      def callAdd() {
9          ScalaTrace.logger.log("8 >>> add$")
10         val addResult = add(8)
11         ScalaTrace.logger.log("add >>> addResult")
12     }
13 }
```

Once the original program is modified into program above, data dependences `add$ >>> x, x 1 >>> add, 8 >>> add$, add >>> addResult` will be passed to analyzing module sequentially when it is executed. It is convenient to insert the "log"'s after the program is parsed into AST.

To explain how to insert the "log"'s into ASTs, first we should know basically how Scala compiler works. Scala compiler has over 20 compilation phases.

Almost all phases take an AST as input, modify the AST and pass it to the next phase.

Our goal is to modify the AST by inserting our "log" operations. Scala compiler plugin is a convenient tool to do this. A compiler plugin is a program written by user that acts as an additional compilation phase. The plugin can be inserted between almost any two standard compilation phases. Just like standard phases, a plugin also takes an AST as input, modify it and produce a new AST, which is passed to next phase.

To insert "log" operations, we should create ASTs for the function calls of `ScalaTrace.logger.log`, then insert them in the appropriate place of the original AST. Following shows a code snippet in our compiler plugin that creates such ASTs.

```
15  def genLog(dataDependence: String, positionInSourceFile: Position): Tree = {
16      def extractEssentialPath(path: String): String = {
17          if(path.contains("NoPosition"))
18              "NoPosition"
19          else
20              path.substring(path.indexOf("/src/") + 4)
21          }
22
23      val liter0 = Literal(Constant(extractEssentialPath(positionInSourceFile.focus.toString)))
24      liter0.setType(typeOf[String])
25
26      val liter1 = Literal(Constant(dataDependence))
27      liter1.setType(typeOf[String])
28
29      val scalatrace = rootMirror.staticModule("ScalaTrace")
30      val logger = definitions.getMember(scalatrace, newTermName("logger"))
31      val callLog = global.gen.mkMethodCall(logger, newTermName("log"), List(liter0, liter1))
32      val Apply(fun, _) = callLog
33      fun.setSymbol(logger.info.decl(newTermName("log")))
34      fun.setType(logger.info.decl(newTermName("log")).tpe)
35      val Select(qual, _) = fun
36      qual.setType(logger.tpe)
37      callLog.setType(typeOf[Unit])
38      callLog
39  }
```

Line 23 and 26 creates ASTs for two string arguments of `log` method. Line 31 creates the function call of `log`, which is an `Apply` AST. The compiler plugin should runs after standard phase `typer`. Because after `typer`, the names and types of ASTs are calculated, so that these can be used in the data dependence information.

However note that AST for `log` is created by us in the plugin, which runs after `typer`. So we have to set the types and symbols for it and its subtrees by ourselves so that it can be a legal tree for the following phases. That's why there are a lot of `setType`, `setSymbol` in the snippet above.

Then our plugin traverses the original AST, calls genLog to create the "log" ASTs and insert them at appropriate places. Following shows the ASTs before and after our plugin phase.

```
class Add extends scala.AnyRef {
  ...
  def add(x: Int): Int = x.+(1);
  def callAdd(): Unit = {
    val addResult: Int = Add.this.add(8);
    ()
  }
}
```

```
class Add extends scala.AnyRef {
  ...
  def add(x: Int): Int = {
    val newvalue = x.+({
      ScalaTrace.this.logger.log("Add.add$ >>>  Add.x", "/print/hello.scala,line-10,offset=160");
      1
    });
    ScalaTrace.this.logger.log("Add.x 1 >>> Add.add", "/print/hello.scala,line-11,offset=191");
  }
  def callAdd(): Unit = {
    ...
    val addResult: Int = Add.this.add({
      ScalaTrace.this.logger.log("8 >>> Add.add$", "/print/hello.scala,line-15,offset=253");
      8
    });
    ScalaTrace.this.logger.log("Add.add >>> Add.addResult", "/print/hello.scala,line-15,offset=238");
    ...
  }
}
```

3.3 Analyzing Data Dependences Extracted from AST

To avoid cyclic dependence when using our tool to analyzing source code of Scala language itself, we implement our runtime analyzing module in Java code. The class **ASTDataFlowGraph** is the main part of our AST analyzing module. Also, we create a data structure to represent source code lines in analyzing module, named **ASTDataFlowGraphNode**. Following shows parts of their source code.

```
public class ASTDataFlowGraph {

    private HashMap<String, ASTDataFlowGraphNode> lineMapToNode = new HashMap<>();
    private HashMap<String, Set<ASTDataFlowGraphNode>> nameLastWrittenBy = new HashMap<>();
    private HashMap<String, Set<ASTDataFlowGraphNode>> backpatchMap = new HashMap<>();

public class ASTDataFlowGraphNode {
    String line;
    Set<ASTDataFlowGraphNode> dependOn = new HashSet<>();
```

To analyze runtime data flow provided by the AST modified with our compiler plugin, we need to keep three maps in our analyzing module, as the code above shows.

1. **LineMapToNode** Maps source code line, in String form, into the data structure used to represent a source code line in analyzing module.

2. **nameLastWrittenBy** For each AST name, this map records in which lines its value is recently updated. For example, suppose the analyzing module receives `log("a >>> b", "line-12")`, then `"b -> line-12"` will be stored into this map.

3. **backpatchMap** The keys of this map are name of values. The values of this map are source code lines which has a data dependence on the key,

collected in a set. This map is used when analyzing module receives data flow `log("a >>> b", "line-12")` but cannot find an item with key `"a"` in **nameLastWrittenBy**. In this case, **ASTDataFlowGraphNode** for `"line-12"` will be put into the set indexed by name `"a"` in this map. Later on, when the value of `"a"` is found updated in some line, then we add this line to the dependence set of every line in the set indexed by `"a"` in this **backpatchMap**. In other words, **backpatchMap** is used to delay the building up of dependence, when the line lastly updates `"a"` is not known. This is quite similar to the idea of *backpatching* when a compiler generates branch instructions.

We define the **log** method in our analyzing module as a static method. Here's the prototype of **log** function.

```
void log(String[] froms, String to, String[] poses) {
```

Here **to** is the name of AST which is executed when this **log** method is called. **poses** are source code lines that this AST covers. **froms** are names of ASTs which determine the value of AST **to**.

Calls to this **log** method are inserted in the AST of the program by our plugin, as Section 3.2 introduces. With three maps above, our analyzing approach can be divided into 4 steps, when this method is called.

1. Find out **ASTDataFlowNode**'s representing lines in **poses**.

```
Set<ASTDataFlowGraphNode> toNodes = new HashSet<>();
for(String pos : poses) {
    if(lineMapToNode.containsKey(pos)) {
        toNodes.add(lineMapToNode.get(pos));
    } else {
        ASTDataFlowGraphNode newNode = new ASTDataFlowGraphNode(pos);
        lineMapToNode.put(pos, newNode);
        toNodes.add(newNode);
```

   This step is simple, we just lookup in the **lineMapToNode** map with each line in **poses**, if there's no corresponding **ASTDataFlowNode**, we'll create one. Then all the **ASTDataFlowNode**'s for lines in **poses** will be stored in **toNodes** set.

2. Find the lines update AST's in **froms** most recently.

```
for(String from : froms) {
    String rawName = extractRawName(from);
    if(nameLastWrittenBy.containsKey(rawName)) {
        for(ASTDataFlowGraphNode node : nameLastWrittenBy.get(rawName))
            fromNodes.add(node);
    } else {
        if(backpatchMap.containsKey(rawName)) {
            for(ASTDataFlowGraphNode toNode : toNodes)
                backpatchMap.get(rawName).add(toNode);
        } else {
            HashSet<ASTDataFlowGraphNode> newBackpatch = new HashSet<>();
            for(ASTDataFlowGraphNode toNode : toNodes) {
                newBackpatch.add(toNode);
            }
            backpatchMap.put(rawName, newBackpatch);
        }
    }
}
```

First for each name in **froms**, we check whether **nameLastWrittenBy** contains the name. If so, extract source code lines where its corresponding AST is updated and put it into **fromNodes**. Otherwise, put lines in **toNodes** into the item for names in **froms** in **backpatchMap**.

3. Add dependences on **fromNodes** to **toNodes**.

```
for(ASTDataFlowGraphNode fromNode : fromNodes) {
    for(ASTDataFlowGraphNode toNode : toNodes) {
        toNode.addDependOn(fromNode);
    }
}
```

4. Update **nameLastWrittenBy** and do *backpatching* if needed.

```
nameLastWrittenBy.put(to, toNodes);
if(backpatchMap.containsKey(to)) {
    for(ASTDataFlowGraphNode toNode : toNodes) {
        for(ASTDataFlowGraphNode toBeBackpatched : backpatchMap.get(to)) {
            toBeBackpatched.addDependOn(toNode);
        }
    }
    backpatchMap.remove(to);
}
```

In this step, we'll put **toNodes** into the item for AST **to** in **nameLastWrittenBy**. In other words, value of AST **to** is lastly updated by lines in **toNodes**. Also, we need to check the **backpatchMap** to see if there's any item with key **to**, if there's such an item in **backpatchMap**, then we set each line in **backpatchMap.get(to)** depends on each line in **toNodes**.

4 ICode Approach

In this section, we introduce our analyzing tool based on ICode. The basic idea of ICode approach is quite similar with AST approach. First we need to insert **log** method calls into the program at compile time. However, instead of insert **log** method calls into AST, we insert them into ICode of the program, which happens in the **icode** phase instead in the compile plugin. Thus this approach requires a modification of **icode** phase of the Scala compiler.

4.1 Represent Data Dependences with ICodes

ICode imitates push/pop operations of a stack machine. Each ICode instruction can first take some values as input from the stack, and then put its result onto the stack. Here's the table of main scala ICodes and their corresponding push/pop operations.

With these operations on the stack, we can know exactly each value on the stack is produced and consumed by which instruction. Thus data dependences can be represented using a sequence of push/pop operations. Again we

[1] push 1 means pushing one value/object onto stack, similarly hereinafter.
[2] n means the number of arguments, similarly hereinafter.

ICode	Operations	Stack Operations
THIS	loads "this" on top of the stack	push 1[1]
CONSTANT	loads a constant on the stack	push 1
LOAD_ARRAY_ITEM	loads an element of an array	pop 2, push 1
LOAD_LOCAL	load a local variable on the stack	push 1
LOAD_FIELD	load a field on the stack	pop 1, push 1
LOAD_MODULE	load a module on the stack	pop 1
STORE_ARRAY_ITEM	store a value into an array at a specified index	pop 3
STORE_LOCAL	store a value into a local variable	pop 1
STORE_FIELD	store a value into a field	pop 2
STORE_THIS	store a value into the 'this' pointer	pop 1
CALL_PRIMITIVE	call a primitive function	pop n^2, push 1
CALL_METHOD	call a method	pop n, push 1
NEW	create a new instance of a class	pop n, push 1
CREATE_ARRAY	create an array	pop n, push 1
SWITCH	a switch instruction	pop 1
CJUMP	jump according to the result of comparing two values	pop 2
CZJUMP	jump according to the result of comparing with zero	pop 1
THROW	throws an exception	pop 1
DROP	drop one value from the stack	pop 1
DUP	duplicate the top of the stack	pop 1, push 2
LOAD_EXCEPTION	load an exception	pop all, push 1

demonstrate with the running example to see how these push/pop operations are extracted from the ICode.

First let's see how the simple `add` function is translated into ICode.

```
5   def add(x: Int (INT)): Int {
6   locals: value x
7   startBlock: 1
8   blocks: [1]
9
10  1:
11    3 LOAD_LOCAL(value x)
12    3 CONSTANT(1)
13    3 CALL_PRIMITIVE(Arithmetic(ADD,INT))
14    3 RETURN(INT)
15
16  }
```

Our simple `add` function is translated into 4 ICodes. First, it loads the parameter `x` onto stack, then a constant `1`. After that, the `ADD` primitive operation is called. Finally, the function returns. The number `3` before these ICodes is the line number of corresponding source code.

Here's the ICode for the assignment `val addResult = add(8)`

```
25    7 THIS(Add)
26    7 CONSTANT(8)
27    7 CALL_METHOD Add.add (dynamic)
28    7 STORE_LOCAL(value addResult)
```

Again it can be represented by 4 lines of ICode. First it loads "this" onto the stack, this is needed to call `add` method. Note that `add` is a method which consumes an object of `Add` class. Since here it is called inside the `Add` class, so "this" object will be used. Then it loads the constant `8`, as the argument for

`add` method. The third ICode calls `add` method. After this ICode, the ICodes of `add` method will be executed. When the `add` call finishes, store its result into `addResult`.

Putting it all together, here's the sequence of ICodes being called when the assignment `val a = add(8)` is executed, with push/pop operations for each ICode shown on the right.

Step	ICode	Operation
1	`THIS(ADD)`	`push @this`
2	`CONSTANT(8)`	`push 8`
3	`CALL_METHOD Add.add (dynamic)`	`pop x, pop @this`
4	`LOAD_LOCAL(value x)`	`push x`
5	`CONSTANT(1)`	`push 1`

6	`CALL_PRIMITIVE(Arithmetic(ADD, INT))`	`pop @operand1`
		`pop @operand2`
		`@operand1 @operand2 >>> @result`
		`push @result`

7	`RETURN`	
8	`CALL_METHOD Add.add (dynamic)`	`push @result`
9	`STORE_LOCAL(value addResult)`	`pop addResult`

In our push/pop operations above, `push <something>` means putting `<something>` on the stack. `pop <something>` means storing the top of stack into `<something>` then pop the stack. Sometimes we need `>>>` to represent data dependences directly. For example `a b >>> c` means the value of `c` is determined by `a` and `b`.

4.2 Obtain Data Dependences at Runtime from ICodes

Now we know how push/pop operations that exactly describes data dependences can be extracted from ICodes. In this section let's focus on how to insert "log" function calls into ICodes.

We modify the icode phase of scala compiler 2.11 to insert additional ICodes for each original ICode. These additional ICodes calls the "log" function, passing push/pop operations to the analyzing unit.

Following shows the ICodes to call "log" function.

```
LOAD_FIELD ScalaTrace.logger
CONSTANT("#1 Add.x @")
CONSTANT("Hello.scala,line-2")
CALL_METHOD Logger.log (dynamic)
```

Our "log" function is a method of Logger class. The program and our analyzing tool will interfere through `ScalaTrace` module, which is a final java class. `logger` is a static field of `ScalaTrace`.

To call `ScalaTrace.logger.log`, `ScalaTrace.logger` should be loaded first. This is done by the first `LOAD_FIELD` ICode. Then two String arguments, line number and push/pop operation, needed by `log` should be put on stack. We can do this using two `CONSTANT` ICodes. Finally, `CALL_METHOD` ICode calls `Logger.log`, using `ScalaTrace.logger` as the object to call this method, with the two String constants as arguments.

For each ICode generated from the original program, these additional ICodes will be added before them, to passing the position and push/pop operations of original ICodes to our analyzing module. Following figure shows ICodes of our `add` function, with additional ICodes.

4.3 Analyze Data Dependences Extracted from ICodes

From section 4.2 we know how push/pop operations and code positions are passed to our analyzing module. This section focus on how to analyze these information are used to analyze data dependences, and find out dead codes.

4.3.1 Build Data Dependence Graph

Here's the basic idea of our analyzing algorithm. Our analyzing modules maintains a stack to imitate every push/pop operation of the program's original ICodes. In this way, whenever an ICode pop something from the stack as an operator, our analyzing module knows exactly which object it gets. For example, at some time, an object `ObjA` is push on the stack in source code position `file1,line-21`. Later on, `ObjA` is popped from the stack in source code position `file2,line-34`. This means `file2,line-34` has a data dependence on `file1,line-21`.

In this way, the data dependences between all lines of source codes in the original programs can be detected. Thus we can build a dependence graph in memory to maintain such relationships. Vertices in the graph corresponds to lines in the source code. Vertices are connected by directed edges. If there's an edge from Vertex A to Vertex B, that means source code line of Vertex B depends on that of Vertex A.

4.3.2 Search for Dead Codes

To search for dead codes within the dependence graph, a target must first be assigned. A target is an variable name in the program that the user considers as the final result of the program. For example, suppose the program is our Scala compiler, then the target can be `jclassBytes`, the variable storing the final java byte code, which is the final product of Scala compiler.

The target should be assigned before the program is executed. Then after the program lanches, our analyzing module will build the dependence graph gradually. Then, when the program is writing the target variable, a push operation `push <target>` will be passed to the analyzing module, along with its source code line. Then vertex corresponding to this source code line will be marked as useful, as well as all descendants of this vertex.

4.3.3 Get the Result

The last question is how to obtain the useless source code lines. Since we only mark source code lines as useful or useless, a straight forward idea is we can simply dump all the source code lines and all the useful source code lines into

two separate files when the compilation finishes, then take the difference of these two files to get the useless lines.

However, to do this the program must be able to inform the analyzing module before it terminates so that the analyzing module can dump the result to files. As the program terminates, all the result in our analyzing module will disappear. This requires special treatment to inserting function calls other than "log" to code segments that the program will execute before terminating. It is extremely hard to identify such code segments at compile time.

Thus whenever a new source code line is executed or being marked as useful, the analyzing module has to write it to the file. Because all the actions of our analyzing module is activated function calls from the program, and it doesn't know when the program will terminate, it has to write the result incrementally.

5 Experiments and Results

We test both approaches on a test set which contains 184 Scala source files. In addition, we test our plugin approach on first two parts of Scala language source code, **library**(585 source files) and **reflect**(157 source files).

On average, plugin approach finds out 65%-75% percent executed code which has an influence on the final output of compiler. With ICode approach, we get 85%. Our plugin did not reach 85% as ICode approach does perhaps due to the point where we insert our compiler plugin. Our plugin runs after typer phase. However, Scala Compiler still does a lot transformation, such as adding some synthesized method. These may cause failure for our plugin to capture the dependence.

We list some source code lines which being detected as dead code by plugin but not by icode approach, and their corresponding source code.

1. `/compiler/scala/tools/nsc/Global.scala,line-596`

```
594   // phaseName = "icode"
595   object genicode extends {
596     val global: Global.this.type = Global.this
597     val runsAfter = List("cleanup")
598     val runsRightAfter = None
599   } with GenICode
```

2. `/compiler/scala/tools/nsc/Global.scala,line-626`

```
622   // phaseName = "constopt"
623   object constantOptimization extends {
624     val global: Global.this.type = Global.this
625     val runsAfter = List("closelim")
626     val runsRightAfter = None
627   } with ConstantOptimization
```

3. `/compiler/scala/tools/nsc/transform/LazyVals.scala,line-14`

```
14    val phaseName: String = "lazyvals"
```

   At first glance, it seems that the version of Scala compiler causes the difference, since the first example deals with `icode` phase, which is not

used in our compiler approach based on Scala-2.12. However, the same case happens in the definitions of other phases in `Global.scala`. This can be caused by defects when we deal with inheritance and mixins. To reach a higher accuracy, our plugin approach needs to identify data flow between members in parent and children classes.

6 Conlusion

Both of our approaches have identify a reasonable amount of code that does not contribute to the java byte class, which we consider as the final output of the compiler. However, our plugin approach performs worse than our icode approach, the problem can be defects when dealing with shared members between parent and children classes. Also, since our plugin runs after the 3rd phase out of the total 25 compiler phases, transformations done by later phases won't be grabbed by our plugin, as well as the data dependences brought by these transformations. Moving plugin to later phases is our next goal, though it may take more efforts since we need to compensate every thing done by phases before our plugin manually.

Though our tools still need some improvement, but the experiment results show that they are practical approaches for dead code detection at runtime. We'll make efforts to enhance them, and hopefully they can be used more widely in the near future.