Making sbt Macro-Aware

Martin Duhem, Eugene Burmako

Technical Report

June 2014

Contents

1 Introduction

Macros made their first appearance as a fork of the Scala compiler and then as an experimental feature in the version 2.10.0 of Scala [1]. Since macros are normal Scala functions that are executed by the compiler during the compilation of their clients, they can introduce dependencies between compilation units in new and exciting ways.

In which ways is post-macro dependency analysis different from its pre-macro days? How do we analyze all the code that's involved in macro expansion, not just the end result? How can we know what parts of a program have been inspected by a given macro in order to produce a specific expansion?

The goal of this report is to explain how macros affect incremental compilation and outline the techniques used to upgrade sbt [2] to provide better support for macro-based programs.

1.1 Incremental Compilation

A complex software project often involves hundreds of classes dispatched among many files. Recompiling the whole project takes a great amount of time, which is why incremental compilers have been developed. Their goal is to recompile only the files that have been modified, and those that are impacted by the newly introduced changes.

To do their work efficiently, incremental compilers need to understand precisely how the different components of the software project are related, and what dependencies exist between them. To do this, sbt analyzes the abstract syntax trees that are output by the Scala compiler after the compilation of a file to determine what are the symbols that are defined in this file (classes, their methods, etc.) and what their dependencies are.

For each file, sbt creates a record containing the path to the file, the symbols that are defined (the file's public `API`), a timestamp and the file's dependencies.

Using this information, sbt can determine whether a change made to a file should trigger the recompilation of another file (that is, *invalidate* this file).

```
object Foo {
  val bar = 42
}
```

```
object Baz {
  def quux = Foo.bar
}
```

Listing 1: Illustrating the dependencies by member reference.

At the beginning of this project, sbt had just changed its invalidation algorithm. The new algorithm, called the Name Hashing Algorithm [3] [4], classifies dependencies in two different buckets : the dependencies introduced by member reference, and the dependencies introduced by inheritance.

Once sbt knows where the dependencies come from, it can decide precisely which files should be invalidated given the changes that have been made.

To illustrate what dependencies by member reference are, please consider the code in Listing 1. In this example, sbt would record that the `object Baz` depends on `object Foo` by member reference, because the method `quux` uses (that is, *references*) the member `bar`. Because sbt recorded this dependency by member reference between `bar` and `Baz`, it knows that, if `bar` is removed from `object Foo` or its type gets changed, then it will need to recompile also `object Baz` to make sure that despite the modifications that have been made to `object Foo`, `object Baz` is still correct.

Listing 2 shows an example where dependencies by inheritance appear. Because `object Baz` extends `class Foo`, sbt will register that `object Baz` depends on `class Foo` by inheritance. With this kind of dependency, if any modification is made to `class Foo`, then sbt knows that it must recompile

```
abstract class Foo {
  def bar: Int
}
```

```
object Baz extends Foo {
  override def bar = 42
  def quux = "hello"
}
```

Listing 2: Illustrating the dependencies by inheritance.

`class Foo`. This is correct, since the changes made to `class Foo` may prevent `object Baz` from compiling no more.

For instance, if we added a method `quux` in `class Foo`, then we would have to make sure that the method `quux` has the `override` modifier in `object Baz`. Moreover, if we added a new abstract method in `class Foo`, we would have to verify that it is implemented in `object Baz`.

1.2 What are the Problems that Arise with Macros ?

First, let us describe rapidly what are the macros in Scala. Macros in Scala are functions that are called by the compiler to inspect and change the program being currently compiled. After the compilation, one cannot tell from the resulting bytecode that a macro has been involved in the generation.

Listing 3, shows one of the simplest macros that can be imagined. During the compilation of `class Bar`, the Scala compiler will see that `Foo.hello` is a macro call, and therefore will *replace* it by the tree that is output by executing the macro. In this case, the call to `Foo.hello` will simply be replaced by the string `Hello, world !`

The bytecode corresponding to the method `func` defined in `class Bar` is shown in listing 4. Please note that the call to `Foo.hello` has been completely replaced, and that no mention of `object Foo` or `hello` appears in the bytecode : the method `func` simply places the string `Hello, world !`

```
object Foo {
  def hello: String = macro impl
  def impl(c: Context): c.Tree = {
    import c.universe._
    val str = "Hello, world !"
    q"$str"
  }
}
```

```
class Bar {
  def func = println(Foo.hello)
}
```

Listing 3: An example of a very simple macro

```
(...)
public void func();
 Code:
  0: getstatic     #16 // Field scala/Predef$.MODULE$:Lscala/Predef$;
  3: ldc           #18 // String Hello, world !
  5: invokevirtual #22 // Method scala/Predef$.println:(Ljava/lang/Object;)V
  8: return
  (...)
```

Listing 4: Bytecode corresponding to `def func`

```
object Provider {
  def withArg(arg: Any): String = macro withArgImpl
  def withArgImpl(c: Context)(arg: c.Tree) = {
    import c.universe._
    val out = "This is defined : " + arg.toString
    q"$out"
  }
}
```

Listing 5: A macro that accepts an argument

on top of the stack, and calls `println` to show this string on the screen.

1.2.1 What Happens to Macro Applications ?

As we have just seen, macro calls get completely replaced by their output. So what will happen to the application of the macro and its arguments if they don't appear in the expanded macro ?

Listing 5 shows an example of macro that accepts an argument, while listing 6 shows an example of how to apply this macro.

The application of this macro will only compile if its argument is defined (otherwise, the compiler would report an error such as `not found: value`

```
object Foo {
  val bar = "Some string"
}
```

```
class Client {
  def func = Provider.withArg(Foo.bar)
}
```

Listing 6: Application of `Provider.withArg`

`something`).

The `AST` (*abstract syntax tree*) that corresponds to `def func` does not contain any mention of the macro, nor of its argument : it will only contain the string `This is defined : Foo.bar`.

What if we removed the member `bar` from `object Foo` ? We should recompile `class Client`, since the expansion of the macro would be outdated ! But how can an incremental compiler know that a dependency between `class Client` and `object Foo` exists ?

1.2.2 Inherited Macros Lead to Spurious Recompilations

Although they look like classical functions, macros behave differently since they replace the calls that are made for them by their output. This is why any change made to the body of a macro must trigger the recompilation of all of its clients, since the output of the macro is likely to be different.

Since sbt tracks changes made to the entire source file rather than to individual parts of it (classes, objects, methods, ...), it doesn't know exactly what part has been modified whenever a change is detected.

Therefore, any change to a source file that defines a macro should invalidate all the clients of this macro, even if the actual body of it has not been modified. But what should be the correct definition of *defines* a macro ?

Considering listings 7 and 8, do we need to recompile `class Bar` whenever a change is made to `object RealProvider` ? The only reason why we would need to recompile `class Bar` would be if a change was made to the body of a macro defined by `object RealProvider`, but `object RealProvider` doesn't define any macros, it only inherits one. Therefore, we know that no matter what changes have been made to `object RealProvider`, we don't

```
abstract class Provider {
  def sayHello: Unit = macro ???
}
```

```
object RealProvider extends Provider {
  val foo = 42
}
```

Listing 7: Extending a class that defines a macro

```
class Bar {
  val prov = RealProvider
}
```

Listing 8: `class Bar` depends on `object RealProvider` by composition

need to recompile `class Bar`, since no changes to a macro body can have been made.

In the first implementation of this invalidation mechanism, sbt considered that any class that provided a macro actually defined one. The problem appeared with classes that inherited macros from other classes and that triggered recompilation of numerous files whenever changes were applied to them [11].

1.2.3 Macros Can Create Dependencies on Arbitrary Symbols

Macros in Scala benefit from the powerful reflection `API` that allows them to inspect any part of the program. Using this `API`, a macro that, for instance, lists all the members of a class can be easily defined. This is exactly what the macro shown in listing 9 does.

Listing 10 shows an example of the application of this macro. During the compilation of `object Client`, the call that is made to the macro `Provider.getMembers[Foo]` will get expanded to a string containing the list of members of `class Foo`.

Anyone would agree with the fact that there exists a dependency relation between `object Client` and `class Foo`. For instance, if a new member was

```
object Provider {
  def getMembers[T]: String = macro getMembersImpl[T]
  def getMembersImpl[T: c.WeakTypeTag](c: Context) = {
    import c.universe._
    val T = weakTypeOf[T]
    val members = T.members.sorted.mkString(", ")
    q"$members"
  }
}
```

Listing 9: Listing the members of a class using a macro

```
class Foo {
  val bar = 42
  def baz = true
}
object Client extends App {
  println("Members of Foo : " + Provider.getMembers[Foo])
}
```

Listing 10: Application of the macro `Provider.getMembers`

added to `class Foo`, we would need to recompile `object Client`, since the expansion of the macro `Provider.getMembers` would be outdated : it would be missing the newly introduced member !

But how can an incremental compiler understand that `object Client` depends on `class Foo` from its `AST` ? All the information that it extracted from `class Foo`, thus introducing the dependency, appears as a simple string in the resulting bytecode.

```
object Provider {
  val baz = 42
  def sayHello: Unit = macro impl
  def impl = ...
}
```

```
class Foo {
  def quux = Provider.baz
}
class Bar {
  def hello = Provider.sayHello
}
```

Listing 11: Modifying `object Provider` causes unnecessary invalidations.

1.2.4 Changing Macro Bodies

Whenever the body of a macro or any helper method that is used in it is changed, we should recompile all the expansions of this macro, because its output is very likely to be different. Therefore, whenever any file defining a macro is modified, sbt recompiles all the files that depend on it, even if the dependency is not related to the macro (sbt only knows that a file has been modified, it doesn't know what part exactly, as explained in 1.2.2). This solution leads to many unnecessary recompilations whenever a non-macro part of a file that defines a macro is modified.

An example of this problem is shown in listing 11 : Modifying the macro in `object Provider` should only invalidate `class Bar`, and adding a new method to `object Provider` should not invalidate anything. At the moment, both `class Foo` and `class Bar` are invalidated whenever `object Provider` is modified, because it defines a macro.

The current solution implemented by sbt does not cover all the cases. Modifying any of the transitive dependencies of the macro implementation should invalidate all expansions of this macro. For instance, in listing 12, changing `val hello` in `object Helper` should trigger the recompilation of all expansions of `Provider.sayHello`, because its output will be different. Unfortunately, at the moment, sbt does not trigger any recompilation in such cases.

```
object Provider {
  def sayHello: Unit = macro impl
  def impl(c: Context) = {
      import c.universe._
    val hello = Helper.hello
    q"$hello"
  }
}
```

```
object Helper {
  val hello = "Hello"
}
class Foo {
  def hello = Provider.sayHello
}
```

Listing 12: Modifying `object Helper` should invalidate the expansion of `Provider.sayHello`.

2 Solving the problems

Now that the problems have been clearly stated, let's explain how these problems have been solved.

2.1 Registering Macro Applications

As stated earlier, during its compilation, the application of a macro is completely replaced by its expansion, therefore erasing some information that we would need in order to fully understand what has been used to produce a specific expansion of the macro.

Fortunately, this information is not completely lost during compilation. Since the development versions of Scala 2.10, the original tree, that is, the one that represents the call to the macro, is *attached* to the expansion of the macro (Scala's compiler infrastructure provides a mechanism to associate collections of custom objects, called attachments, with abstract syntax trees, and the macro engine makes use of it).

This tree can therefore be retrieved by looking up the attachment from the expansion of the macro and can be analyzed. The goal of this analysis is to get the name of the macro and the arguments of the call. Having the macro and the arguments that were passed to it allows sbt to register new

dependency relations between the macro client and the macro provider, and also between the macro client and the elements that have been passed as arguments to the macro.

The actual fix to the problem required to retrieve and walk these original trees, and insert the extracted data into sbt's dependency tracking system. Along with this work, we also worked on improving the unit testing framework related to this part of sbt, to make it able to compile macros and their applications.

One of the great difficulties of this work was to extract the original trees in a way that would not reduce the overall performance of sbt. Since sbt must stay source compatible with previous versions of the Scala compiler (from version 2.8 up to the latest), we encountered a problem in extracting the original trees, because tree attachments simply did not exist in versions of Scala prior to 2.10.0, and because the attachment we were interested in (`MacroExpansionAttachment`) has been moved and modified between Scala 2.10 and 2.11.

In order for a reference to `MacroExpansionAttachment` (which only exists in Scala 2.10 and 2.11) to compile with earlier versions of Scala, one needs to stub it out somehow (an alternative would be to use structural types, but that would significantly degrade performance). However, stubbing has to be done carefully in order not to shadow the actual `MacroExpansionAttachment` when compiling with Scala 2.10 and 2.11.

To overcome this difficulty, we used a subtlety of Scala : imports can be placed anywhere, and their scope is limited to the current block. Moreover, the priority of imports depends on the level of nesting. The problem was therefore solved by creating a dummy attachment data structure, so that versions of Scala that don't have the required attachment can use this substitute. Next, we needed to import the real attachment for versions of Scala that offer it. In the Scala 2.10 series, this attachment can be obtained directly by importing the content of `Global`. In Scala 2.11, it can be found in `Global.analyzer`. To make the correct imports without having ambiguous names (a name that have been imported twice in the same scope), we nested `locally { }` blocks : the compiler will use the innermost definition

```
// Stub MacroExpansionAttachment for Scala < 2.10.x
object stubs {
  class MacroExpansionAttachment { ... }
}
import stubs._
locally {
  import global._ // Real MacroExpansionAttachment for Scala 2.10.x
  locally {
    import analyzer._ // Real MacroExpansionAttachment for Scala 2.11.x
    // Use MacroExpansionAttachment
  }
}
```

Listing 13: Dealing with previous versions of Scala

or import of the `MacroExpansionAttachment`.

Please consider listing 13, which presents a simplified version of the solution used to circumvent this problem. Imagine that we are interested in using the `class MacroExpansionAttachment`, that does not exist in versions of Scala prior to 2.10, is defined in `global` in Scala 2.10, and in `global.analyzer` in Scala 2.11. Note the use of wildcard imports : we cannot simply `import global.MacroExpansionAttachment`, since this won't be defined for versions of Scala different from 2.10. The full solution to this problem can be found in reference [5].

The original fix for the registration of macro applications was supposed to be included in sbt version 0.13.2 [6], but was reverted because of the discovery of a bug due to cyclic chains of *original trees* [7] [8].

After a workaround for this bug had been added, this fix was introduced in sbt version 0.13.5 [9] [10].

2.2 Handling Inherited Macros

To properly handle inherited macros and in order to reason efficiently about whether or not a source file *defines* a macro, we needed to apply some fixes to the way sbt creates its internal representation of the `API` of a source file.

To decide if a source file defines a macro, sbt walked all the members of a class and, if it discovered a member being a macro, then it considered that the file defined one.

While this approach is correct, the reason why spurious recompilations were observed with inherited macros comes from the fact that sbt also considered inherited members while looking for macro definitions. Therefore, it considered that classes which simply inherited a macro from another class defined a macro, and therefore triggered the recompilation of a large number of files whenever a change was applied to these classes.

For instance, this problem led to a huge number of recompilations with big projects. For instance, a whitespace change to `Global` in the Scala compiler, which inherits macros from other classes, triggered the recompilation of hundreds of files that depend on it [12].

The fix to this problem [13] has been released with sbt 0.13.5 [9] [10].

2.3 Registering the Dependencies of Macro Expansions

Registering the dependencies of macro expansions is a more elaborate problem, since we cannot base ourselves on the abstract syntax trees that are generated by the Scala compiler : they could not contain any information relevant to understand the relationships that exist between the different files of a project.

To address this problem, we implemented a compiler plugin [14] whose job is to register the calls that are made to the reflection `API` among with the symbols that are retrieved (*touched*) using it.

The plugin makes use of the MacroPlugin infrastructure available in Scala 2.11.0 to hook into the workflow of macro expansion. After being registered in the macro engine, the plugin wraps standard macro context objects (contexts provide reflection `API`s to macro implementations) into proxies that register

all calls being made and recursively create proxies for all data structures that get passed between the compiler and macros. Such proxies can then keep track of the reflection `API` calls, for example registering touched symbols.

This list of all touched symbols is then attached to the expansion of the macro, and can therefore be retrieved by incremental compilers and be used to reason about invalidation of files.

Retrieving this list in sbt was actually much more complicated than expected. At first, we attached this list as a `case class`, which required us to write a library that we could use in sbt to retrieve this attachment. Unfortunately, this part of sbt cannot depend on any library [17].

As a solution to this problem, we decided to store the touched symbols as a `Map[String, Any]`. The touched symbols can be retrieved by looking up `touchedSymbols` in the map and casting the result to `List[Symbol]`.

This compiler plugin made it possible to track the dependencies of macro expansions and also solved the problem that is shown in listings 9 and 10 in sbt.

Unfortunately, there exists no infrastructure to store dependencies that come from a macro expansion in the current implementation of sbt, and we needed to add a new "bucket" to hold dependencies coming from a macro expansion.

Adding this new infrastructure so that sbt can understand this new kind of dependency means changing the signature of many methods and modifying many parts of sbt. Since the number of kinds of dependencies is likely to grow, we decided that refactoring the way sbt stores dependencies was necessary [15].

A first integration in sbt of the macrotracker compiler plugin can be found in reference [16].

3 Refactoring sbt's Dependency Tracking System

The discussion to merge support for the macrotracker compiler plugin in sbt led to a discussion of the refactoring of the way dependencies are registered.

```
def addExternalDep(src: File,
  dependsOn: String,
  inherited: Boolean): Relations
```

Listing 14: Original signature

```
def addExternalDep(src: File,
  dependsOn: String,
  context: DependencyContext): Relations
```

Listing 15: Refactored signature

Signature of a method used to register dependencies

In the current implementation, a dependency is passed to the functions that are responsible of registering it, among with a simple `boolean` value that indicates whether the dependency comes from an inheritance relation or from a member reference, so that sbt knows where to register the dependency.

Unfortunately, adding a `boolean` parameter to these functions in order to indicate whether the dependency comes from a macro expansion doesn't really make sense, since one would be able to register dependencies which come both from a macro expansion *and* inheritance. Moreover, each time a new kind of dependency is added, the signatures of many methods would need to be modified.

It has been decided to define a few *dependency contexts* that represent the origin of the dependency. Listing 14 shows the signature of an example method before the refactoring, listing 15 shows the signature of the same method using the new dependency contexts.

This new abstraction allows sbt developers to easily add new kinds of dependency and to implement all the relevant invalidation logic without having to modify a huge amount of files and keeping a lot of deprecated methods whenever a new dependency context is added.

The work to refactor sbt's dependencies is still in progress [18], but will come in handy when integrating future support of the macrotracker compiler plugin or to add any other kind of dependency context.

4 Conclusion

As we've seen, macros bring quite a number of changes to the previously established scheme of tracking dependencies between program elements in sbt.

First, in macro-enabled programs, code can disappear into thin air, becoming invisible for traditional dependency analysis. Second, in presence of macros traditional handling of inheritance becomes too imprecise. Also, during their execution, macros can inspect arbitrary program elements, creating unpredictable dependencies. Finally, changes made to macro bodies or any of their transitive dependencies may impact their expansions.

Over the course of the semester project, we've fully addressed the first two problems, with our solutions being available in the upcoming production version of sbt 0.13.5. We have also prototyped a fix to the third problem, but its production implementation ended up requiring a major refactoring of sbt's internal infrastructure. The fourth problem, regarding changes to macro body and their transitive dependencies, is left for future work. At the moment of writing we're in the middle of the refactoring and are looking forward to making our developments available to sbt in the coming future.

5 References

1. http://www.scala-lang.org/download/changelog.html#macros
2. http://www.scala-sbt.org
3. https://github.com/sbt/sbt/pull/1042
4. https://github.com/gkossakowski/sbt/wiki/Incremental-compiler-notes#name-hashing-algorithm
5. https://github.com/sbt/sbt/blob/0.13/compile/interface/src/main/scala/xsbt/Compat.scala#L96
6. https://github.com/sbt/sbt/pull/1163
7. https://github.com/sbt/sbt/issues/1237
8. https://issues.scala-lang.org/browse/SI-8486
9. https://github.com/sbt/sbt/compare/0.13.2b...0.13.5
10. https://github.com/sbt/sbt/releases/tag/v0.13.5
11. https://github.com/sbt/sbt/issues/1142
12. https://groups.google.com/d/topic/scala-internals/STmaXmH2RQg/discussion
13. https://github.com/sbt/sbt/pull/1202
14. https://github.com/scalamacros/macrotracker
15. https://groups.google.com/d/topic/sbt-dev/clIbzCp9VX8/discussion
16. https://github.com/Duhemm/sbt/tree/sbt-with-scalahost-2
17. https://groups.google.com/d/topic/sbt-dev/4jjt-0PBZLc/discussion
18. https://github.com/sbt/sbt/pull/1340