Macros in sbt: Problem solved!

Martin Duhem, Eugene Burmako

Technical Report

January 2015

Contents

1 Introduction

Between February and June 2014, we described [1] how incremental compilation was made more complicated when macro-enabled programs are involved, and we implemented the foundations of the support for metaprograms in sbt, a build tool and incremental compiler for Scala.

From June to September 2014, we worked on improving the internal representation of dependency relationships between files in sbt, to make it easier to extend sbt and to define new relationships.

Starting in September 2014, we benefited of this new infrastructure to fix all the remaining problems that sbt had with macros: how should we handle their transitive dependencies? How can we know what they inspect during their expansion? Are there other means by which macros could introduce dependencies? How should we use these informations?

In this report we will expose the new techniques that have been proposed and implemented to offer a complete support for metaprograms along with all their dependencies in sbt, and explain the most relevant parts of their implementation.

1.1 What problems do macros bring?

In our previous report [1], we presented some of the reasons why incremental compilation is hard to perform correctly when metaprograms are involved, along with our solutions to overcome these problems. Some of the problems that were left for future work, have been addressed now.

1.1.1 The problems we addressed in our previous project

As a reminder, let us present briefly the most challenging problems that are brought by macros, those that we have already solved during our previous project, and the solutions that we proposed at that time.

Dependencies of macro applications When a macro is expanded, its application is completely replaced by its expansion. However, the macro application may have dependencies (for instance, if the macro application takes

parameters). Since sbt couldn't see the macro before its expansion, these dependencies were lost and this led to inconsistencies. For instance, a macro client could give a macro an argument `Foo.bar`. If `Foo.bar` is modified, we should obviously recompile the macro application, even if `Foo.bar` does not appear at all in the expansion.

The solution to this problem is to use a special *attachment* that comes with the macro after its expansion: This attachment contains the macro application before expansion, that is, the original tree. By inspecting this attachment, we were able to determine what are the dependencies of the macro application, and then to register them.

Macros can create dependencies on arbitrary symbols Since metaprograms have access to the reflection API, they can inspect any part of the program, and thus introduce new dependencies. For instance, a metaprogram could look up the list of methods defined by a class, format it as a string and output it. From the expansion of the macro, which would simply be a string, sbt couldn't know that, whenever a new method is added or removed from the inspected class, it has to recompile all the expansions of this macro.

The solution that we proposed to this problem takes the shape of a compiler plugin, called Macrotracker [2], that will attach to the expanded macro the reference of all *symbols* that have been inspected in order to produce a given expansion.

This plugin simply acts as a proxy for the reflection API and registers all calls made and the results that are returned. When the macro expansion is finished, the plugin attaches to the expansion the list of touched symbols to the expanded macro. Then, sbt can extract these symbols and, whenever one of them is modified, it recompiles the macro client.

Even if the compiler plugin is ready since May 2014, sbt doesn't support it yet, because sbt's internal infrastructure needed some improvements that were not yet implemented at that time. Now that the relevant parts of sbt's infrastructure have been improved, we've proposed an implementation in sbt that is being reviewed as of December 2014, and whose details are explained in section 4.3. We will give more details regarding the refactoring that we

```
object Provider {
  def getConfig(file: String): String = macro impl
  def impl(c: Context)(file: c.Tree): c.Tree = {
    import c.universe._
    file match {
      case q"${name: String}" =>
        val content = scala.io.Source.fromFile(name).mkString
        q"$content"
      case _ => c.abort(file.pos, "Not a literal string")
    }
  }
}
```

Listing 1: A macro that loads a value from a file.

undertook in section 3.

1.1.2 New problems that have been tackled

Now that the kind of challenges brought by macros are clear in our minds, let's present the new problems that we worked on fixing.

Allow macro clients to depend on auxiliary (non-scala) files As a compile-time facility, macros are particularly well suited for code generation. However, code generation may very well rely on external resources, such as files whose content is not Scala code.

For instance, the macro presented in Listing 1 could be used to load the configuration of an object at compile time (it may very well load an API key for some service from an external text file). Obviously, whenever one of the external files that are used in this fashion is modified, we need to recompile all the macro clients that make use out of it. How can we extract this information? How can we tell sbt that a Scala source depends on an external, non-scala file?

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

```
object Helper {
  def sayHello = "Hello " + Signs.BANG
}

object Signs {
  val BANG = "!"
}
```

Listing 2: Transitive dependencies of macro implementations

Account for the transitive dependencies of macro implementations Since macro applications get completely replaced by their expansion during compilation, the transitive dependencies of the macro implementations must all be considered to know whether a macro expansion is outdated or not.

For instance, consider the very simple macro presented in Listing 2. If a change was made to `Helper.sayHello`, we would have to recompile all expansions of this macro. This is easy to see, because `object Provider` explicitly references `Helper.sayHello`. However, we would also have to recompile all expansions if `value BANG` in `object Signs` was to be changed, because the expansions would not match anymore what a fresh compilation could give us.

To solve this problem, we have to review the way we think about dependencies and invalidations: usually, when we change the body of a function, we are only interested in recompiling the file that defines it, and not the other files that use it. Unfortunately, sbt's incremental compiler has been developed with this idea, since macros didn't exist in Scala at this time.

Our solution to this much more complex problem is presented in section 4.2.

2 How does sbt's incremental compiler work?

Rather than explaining again the classification of dependencies that sbt operates while traversing the compiled trees like we did in our previous report [1], let us get in more details regarding the internals of sbt's incremental compiler.

During the compilation of each file of a project, sbt collects informations about it: its dependencies, the symbols it defines, their APIs, and a timestamp that sbt will use to know whether the file has been modified or not since the last time it has seen it. This timestamp correspond either to the last time that the file has been modified, or to a hash of the file.

To collect these informations, sbt adds three new phases in the Scala compilation pipeline. After the `typer` phase, sbt injects a new phase called `API`. The goal of this phase is to create a representation of the public API of each class. This representation is then stored in an instance of an object called `AnalysisCallback`, which is essentially a buffer that contains all the information collected during a compilation.

Right after the `API` phase comes another phase added by sbt that is called the `Dependency` phase. The goal of this phase is to walk the trees and collect all symbols that are used in every file. These symbols are then mapped to their definition file, and sbt can register the dependencies between files.

At this point, sbt separates dependencies between files in the same subproject (*internal dependencies*) from cross-project dependencies (*external dependencies*). Whenever sbt encounters an external dependency, it retrieves the API of the file depended on at this moment and stores it in the `AnalysisCallback`.

Once the compilation is finished, the `AnalysisCallback` contains all the information that was extracted during the compilation. Sbt then takes this information and creates an `Analysis` with it. This object contains all the information that sbt needs to know about a project to do its job:

1. All the sources it contains and their APIs
2. All the dependency relations
3. A timestamp for every file (called `Stamp`)

4. The APIs of every external dependency

Now, each time that the user issues the `compile` command in sbt, sbt will compute the set of files that it needs to recompile. First, it will look for source files and `JAR`s that have been modified by comparing the previous value of their `Stamp` with a freshly computed `Stamp`. If a source file has been modified, it will be recompiled. If a binary dependency has been changed, then all the files that use it will be recompiled.

After an internal source file has been recompiled, sbt will check whether its new API is different from the API it previously computed. In which case, it will recur and invalidate more files based on the changes to the API and the invalidation algorithm that is used. For external dependencies, the story is not different. Sbt compares the last known API of every external dependency with its current API, and performs invalidation based on these changes and the invalidation algorithm.

Note that the distinction between internal and external dependencies is not the only classification of dependencies that sbt performs. Sbt also groups dependencies by their *context*. The context of introduction of a dependency represents the kind of dependency relationship that exists between the two files, and allows us to perform a more finely grained invalidation based on a set of changes. The two major dependency contexts that exist in sbt are the dependencies by member reference, and those by inheritance.

In the next section, we'll present the work that we've done to make this second dependency classification more easily extensible.

3 Improvements for dependency management

Developing a new and improved way to store dependency relations between files in sbt allowed us to produce new and exciting prototypes that could handle new kinds of dependencies without applying further changes to the public API of sbt's dependency tracking system.

Prior to our work, the interface that permitted to register new dependency relations was closely tied to the only two dependency relations that sbt considered: dependencies by member reference or inheritance.

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

```
object Helper {
  val hello = "Hello"
}
```

Listing 3: Provider.scala depends on Helper by *member reference*.

These only two kinds of dependency relations don't fit the needs of macro-aware incremental compilation: sbt's incremental compiler has been designed to perform just as many invalidations as required to provide a consistent binary form of a program.

However, when macros are involved, we may need to perform invalidations that wouldn't be necessary with classic programs. For instance, let us consider the macro presented in Listing 3.

In this example, `object Provider` references the member `value hello` of `object Helper`. If the value `Helper.hello` is modified, we should recompile all the expansions of the macro `sayHello`. However, a classic Scala program that uses this value shouldn't be invalidated. It becomes clear that we need to differentiate between dependencies that are introduced in a macro implementation from other dependencies[1].

Another example is dependencies that are collected by the Macrotracker compiler plugin. Consider the macro presented in Listing 4. By looking at the code of the macro implementation, we quickly understand that we won't be able to tell what members of the inspected type have been referenced to produce a given expansion. Still, whenever a type is modified, we will need to recompile all macro clients that inspect it using this macro. This

[1]Our solution to this problem, known as the transitive dependencies of macro implementations, is presented in section 4.2

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

Listing 4: Macro listing the members of its type argument.

means that, if members that didn't even exist at this time are added to the inspected time, we will need to perform invalidations. Since this requirement wasn't necessary in Listing 3, we may consider this kind of dependency as a new one[2].

Our solution to make it easier to add new dependency kinds in sbt relies on an abstraction over the dependency kinds and is threefold: first, we implemented it for the actual representation of *relations* in sbt, then for the part that is in charge of persisting the relations to disk, and finally to the interface between the Scala compiler and sbt.

This work has been implemented with the goal of making it quick and easy to add new dependency kinds in sbt. To add a new dependency kind, we now only have to define the new dependency context, register the new relation and implement the extraction and invalidation logic. Persisting the new relation will work out of the box, and sbt will automatically sort the results of the analysis based on the informations that have been extracted during compilation.

The improvements that this refactoring brought allowed us to produce rapidly new prototypes without having to change many public APIs and thus breaking binary compatibility, which in turn made it both easier and

[2]The details of our implementation to solve this problem can be found in section 4.3

faster for the result of our experiments to reach production quality.

The discussion that accompanied the refactoring along with comments about the code that has been produced can be found in [3], [4] and [5].

4 Details of our solutions

Thanks to the improvements described in the previous section, sbt can easily support new kinds of dependency relations between files. Let us now explain the details of the implementations that we worked on, and how they solved the problems that we presented earlier.

4.1 Registering dependencies on auxiliary files

As we've seen, a macro implementation may very well use some external files to produce its expansion, and the clients of this macro should therefore be recompiled whenever an auxiliary file that has been used in such fashion is modified. The idea for this addition came from a discussion on the *scala-internals* mailing list [6].

An incremental compiler is originally built to deal with code, not with arbitrary files whose content may be completely unintelligible to it, so how can we make sbt handle auxiliary files?

There is already a mechanism in sbt that is in charge of detecting changes to various files. Remember that whenever sbt encounters a new file, it computes a `Stamp` that corresponds either to the hash of the file or to the last time the file has been modified. Using this information, sbt is able to detect whether or not a file has been modified.

However, making sbt handle dependencies on auxiliary files is not quite as simple as that, because we do not know in advance where these auxiliary files will be: the auxiliary files that have been used in a compilation unit must be explicitly given to sbt.

Sbt won't actively look for these auxiliary files, because they may be anywhere on disk. Therefore, if an auxiliary file is no longer advertised, sbt would normally consider it deleted even though the file is still present and

```
def impl(c: Context)(file: c.Tree): c.Tree = { import c.universe._, compat._
  file match {
    case q"${name: String}" =>
      val file = new java.io.File(name)
      val content = scala.io.Source.fromFile(name).mkString
      val map = new java.util.HashMap[String, Any]
      map.put("touchedFiles", file :: Nil)
      val res = q"$content"
      res.updateAttachment(map)
    case _ => c.abort(file.pos, "Not a literal string")
  }
}
```

Listing 5: Improving Listing 1 by attaching the list of touched files

unchanged. To overcome this problem, we had to make sbt copy all the `Stamp`s it has regarding auxiliary files between compilation runs.

Finally, how does sbt know that a particular expansion uses an auxiliary file? Currently, our implementation requires the macro author to attach to the expansion a `java.util.HashMap[String, Any]` that contains a mapping from `"touchedFiles"` to a `List[File]`. This list must contain all the files that have been used to produce this expansion. An example of macro that attaches the list of touched files to its expansion is showed in Listing 5.

Attaching the list of touched files to the expansion of the macro is a burden that is currently left to the programmer. However, having a standardized API to access external files[3] from macro implementations, we could easily leave this to the Macrotracker compiler plugin. Support for dependencies on auxiliary files by sbt would then be completely invisible to the macro authors.

Once sbt encounters a tree that contains this attachment, it simply extracts it and registers the list of external files along with the other depen-

[3]The `resources API` will be in charge of this in `scala.meta` in the future.

dencies.

Obviously, the invalidation logic for this kind of dependency is very simple, since we cannot really reason about the content of the file. Whenever a file that is used as an auxiliary dependency is modified, all the files that depend on it are recompiled.

We have proposed to integrate this work in sbt. Even though sbt developers and the community seemed enthusiastic about this change, its integration will unfortunately have to wait until binary compatibility between versions of sbt can be broken, because this change requires a modification of a few public APIs. This discussion can be found in [7].

4.2 Transitive dependencies of macro implementations

As we said before, macro expansions are very fragile because the result of a macro expansion gets inlined and completely replaces the macro application: a small change to a part of a program that may seem unrelated may lead to a completely different expansion.

Therefore, we need to track all the transitive dependencies of macro implementations, and make sure that whenever any of these dependencies gets modified, we recompile all the clients that use this macro.

We need to be able to differentiate between dependencies introduced in the body of a macro implementation and the dependencies introduced anywhere else in a Scala program. Luckily, we can now easily define a new dependency relation, that we called `DependencyFromMacroImpl`. We also need a way to determine whether a method declaration may be used as a macro implementation. There are two different situations in which a method can be used as a macro implementation:

1. The method must have an argument that can be used as a `Context`, or
2. The method must be part of a macro bundle[4].

[4]We detect macro bundles by checking whether the primary class constructor has a parameter that can be used as a `Context`.

This heuristic might give rise to false positive, but unfortunately we cannot simply collect the methods that are used with the `macro` keyword and register their dependencies, because a method that *may* be used as a macro implementation could be used as such by a different object, in a different file or project: we would miss many macro implementations.

Being able to detect macro implementations and classic Scala functions, we can now collect all the dependencies that are introduced from within a macro implementation, using the same techniques that sbt currently does for other kinds of dependencies (traversing the compiled trees). This relation will be used as the basis to determine whether clients of a macro provider should be invalidated given a set of changes. Here's how we will proceed to compute the set of macro providers that are impacted by a given set of changes:

1. Let M be the set of files that implement macros.
2. Let D_D^m be the set of files that macro implementation m depends on by `DependencyFromMacroImpl`.
3. Let D_f be the set of files depending (directly or transitively) on file f:

$$D_f = \{x : x \text{ depends on } f \vee x \text{ depends on } f', f' \in D_f\}$$

4. Given a set $C = \{c_1, \ldots, c_n\}$ of files that have been changed, we know what macros are impacted:

$$I_M = \{m : m \in M \wedge |D_D^m \cap \bigcup_{f \in C} D_f| \geq 1\}$$

5. We can now invalidate all macro providers that are impacted by the set of changes.

This algorithm seems very simple in theory. However, applying it to sbt's incremental compiler is slightly more complicated because of the separation of *internal* and *external* dependencies.

The result of this separation is that we cannot compute the set D_f for every file f so easily: if this file has external dependencies, then we will need

to retrieve the corresponding `Analysis` that contains informations about this file and its dependencies. This makes the algorithm slightly more complex, but it shouldn't be hard to implement[5].

Since we didn't know how to fetch the `Analysis` corresponding to a given file during our experiments, we implemented a temporary solution that modified the results that sbt provides when detecting changes to an API: If a file, or anything it depends on, has been recompiled, then we made sbt consider all relevant macro providers for invalidation.

Unfortunately, a shortcut mechanism that is used to speed up the invalidation made everything more complicated and we had to actually recompile all transitive dependencies of macro implementations to be able to issue all signals saying that a file has been recompiled but its API has not changed. Therefore, our solution performed many unnecessary invalidations, but was able to recompile macro implementations whenever needed.

Now that we have a solution to load the required `Analysis`, we will be able to implement a solution that won't require all these invalidations. However, this implementation is left for future work, and should be implemented very soon.

4.3 Supporting Macrotracker in sbt

Bringing support for Macrotracker in sbt was the original reason why we started refactoring sbt's dependency tracking system. Once the refactoring was finished, we were finally able to implement a short and elegant solution to make sbt account for dependencies introduced by calls to the reflection API.

In order to implement support for Macrotracker, we needed first to extract the information that Macrotracker gives us from the compiled trees. This was done by creating a new *traverser* that walks the trees and collects the results of Macrotracker's work, a `Map` that contains a mapping from `touchedSymbols` to the list of symbols that have been inspected during the expansion, whenever it encounters it. The dependency between the macro client and the

[5]This solution has not been implemented at the time of writing.

symbols that are inspected is called `DependencyByMacroExpansion`.

Then, we had to implement the complete invalidation logic, so that the macro applications are recompiled whenever an inspected symbol is modified.

The invalidation logic works as follows:

- Whenever an internal source is changed, we invalidate all the files that have a `DependencyByMacroExpansion` on it. If a file depends by macro expansion on another file, then it means that it inspected one of the symbols it defined.
- Whenever an external source is changed, we get the set of internal source files that have a `DependencyByMacroExpansion` on this file, and invalidate them.

The support for Macrotracker has not been merged yet in sbt. However, its integration is currently in discussion [8].

5 Conclusion

This semester project has been an opportunity for us to address all the problems that were left at the end of our previous project.

First, we finished the refactoring that we started during our previous semester project. Refactoring sbt's internal representation of dependency relations really provided us with a very modular version of sbt that allowed us to rapidly produce new prototypes that suited our needs. We are convinced that the improvements brought by this refactoring will benefit other projects that could create new kinds of dependencies.

Then we invested our efforts in making sbt support auxiliary dependencies on external files, an idea that emerged from the *scala-internals* mailing list. Being able to depend on auxiliary files is an interesting idea for many projects that require generating Scala trees from external files. For instance, Scalatex [9] is a programmable document generator in Scala that could benefit from this addition, as soon as it gets merged.

We also implemented support for Macrotracker in sbt, which will hopefully be merged soon in sbt's codebase. The refactoring that we brought to sbt allowed us to provide a concise implementation.

Finally, we worked on making sbt account for the complete transitive dependencies of macro implementations. This problem pushed us to implementing new invalidation techniques and to performing many experimentations. At the time of writing, we do not have a complete implementation of the solution proposed in section 4.2, but we already have a prototype that works as expected.

Even if our modifications have not been distributed yet in an official release of sbt, they have been well received by the sbt team which suggests a possible integration of our contributions in a future release of sbt.

Once our work gets released in an upcoming version of sbt, metaprogramming aficionados will be able to enjoy full-fledged support for macro-enabled programs in sbt.

References

[1] Martin Nicolas Duhem and Eugene Burmako. Making sbt Macro-Aware. Technical report, 2014.

[2] Martin Nicolas Duhem and Eugene Burmako. MacroTracker. https://github.com/scalamacros/macrotracker, 2014.

[3] Abstract over dependency kind in Analysis. https://github.com/sbt/sbt/pull/1340, 2014.

[4] Don't hardcode existing relations in TextAnalysisFormat. https://github.com/sbt/sbt/pull/1572, 2014.

[5] Abstract over dependency kind in Compile. https://github.com/sbt/sbt/pull/1736, 2014.

[6] Best way to provide extra trees to the compiler? https://groups.google.com/forum/#!topic/scala-internals/LcwD6SVuqms.

[7] Registering dependencies on auxiliary (non-scala) files. https://github.com/sbt/sbt/pull/1757, 2014.

[8] MacroTracker support. https://github.com/sbt/sbt/pull/1778, 2014.

[9] Li Haoyi. Scalatex - Programmable documents in Scala. http://lihaoyi.github.io/Scalatex, 2014.