Project Report

Linker call graph with full Dotty support

Nicolas Stucki
Supervisor: Martin Odersky

LAMP
École Polytechnique Fédérale de Lausanne

January 30, 2017

1 Overview

This project [1] consisted in bringing the *linker* [10] call graph up to date with *dotty/master* (section 2), restructure the code (3), make it support the full Scala language [9, 5] (section 4.1) and make it support calls to Java code [4] (section 4.2). Also implemented dead code elimination of unreachable methods based on the call graph [7] as a first optimization (section 3.3). Section 4.3 describes a new graph visualization.

2 Making it stable

The firsts tasks in this project was to make the *linker* [2] call graph construction work on top of *dotty/master* [3]. This consisted in updating to different tree and type shapes that come from the typer. The first iteration was the most complicated one as the linker was forked quite a long time ago. After this first rebase, subsequent rebases have been becoming simpler and stabler (with the exception of the new higher kinds encoding).

3 Architecture

Once the call graph construction was stable enough, I started working on the architecture of the code to make it simpler to understand and easier to maintain. The original code was a prototype and as such was monolithic. The current version consists of a two phase for the call graph construction *CollectSummaries* and *BuildCallGraph* and one for the dead code elimination *DeadCodeElimination*.

3.1 Method summary collection

CollectSummaries is a phase that creates *MethodSummary*s which are summaries of all the calls effectuated in a particular method body. The main infor-

mation in the *MethodSummary* is a map from call receivers to a list of *CallInfo*. These *CallInfo* contain a *TermRef* reference to the method called along with the types of the arguments passed in this call. *CollectSummaries* will also materialise some calls that are not yet present in the trees (see 4.1).

3.2 Building the call graph

BuildCallGraph is a phase in the pipeline that runs after all method summaries have been collected in *CollectSummaries*. This consists in registering all the entry points of the program into a *CallGraphBuilder* and then let the it build the complete graph from these entry points. Currently entry points consist of all *main* methods and all methods annotated with *@EntryPoint* with their class/-module initialisers. The *CallGraphBuilder* is the core and contains the logic to build the call graph. This builds a graph on nodes defined by *CallInfoWithContext* which are *CallInfo* that also has the outer type parameters of the current call. Edges of the graph are listed in each *CallInfoWithContext*. We wont go into details of how the call graph is expanded while building as the details are too long for this report.

3.3 Deadcode elimination

The *DeadCodeElimination* is a phase that runs on all methods that can be *DCEed* that have been materialised. In this phase bodies of unreachable methods are replaced by a *throw new DeadCodeEliminated*. We also wish to eliminate classes that are never used but this is currently not done as it makes error messages from classes loaded through reflection less explicit on the source of the issue. But classes that could be eliminated have only methods that have the same body that throws which makes them efficiently compressible.

3.4 Test infrastructure

To test the call graph and the dead code elimination we need some custom test infrastructure. We currently have two sets of tests that run on top *partest*. The first set of tests that dead code eliminate on the whole Scala standard library with the test code. The other set of tests only compiles the test code. Tests using the Scala standard library compile a copy of the *stdlib* along with the tests files. These are quite slow as the whole compilation pipeline is run on the complete *stdlib*, but doing so cover most of the Scala language features while constructing the call graph. Unsurprisingly from these tests came most of the bugs that were found during this project, including bugs on *dotty/master*. The ones that do not use the standard library, use the precompiled Scala library and handles calls to it as if they are calls on methods defined in java bytecode (see 4.2). These tests are fast and are used to tests single language features and to avoid regressions.

3.5 Future infrastructure work

To support separate compilation, we need to be able to save method *MethodSummary* in the *tasty* file and load them when needed. The code to does it exist but still needs to be updated to the current *dotty/master*.

4 Full Scala support

4.1 Adding missing calls

As *CollectSummaries* runs as early as possible in the compilation pipeline, there are some calls that do not exist yet in the trees. These calls need to be synthesised and added to the *MethodSummary*s.

4.1.1 Mixin constructors

The constructor of some class that uses mixins could have a call to the initialiser of the mixed in trait (if it has one). Therefore when there we call a class primary constructor we add all directly inherited mixin initializers.

4.1.2 Varargs

When we call a method with a variable number of arguments these will be converted to a *WrappedArray[T]*. Hence we need to synthesise a call to one of *genericWrapArray*, *wrapRefArray*, *wrapIntArray*, *wrapLongArray*, ..., *wrapUnitArray*. On the other side, in the method called, we need to adapt the type of the arguments corresponding *WrappedArray[T]*.

4.1.3 Predef module load

At this phase calls to methods on the *Predef* module are missing the call to the module getter. These can be in a variety of different shapes including calls to *LowPriorityImplicits*, *DeprecatedPredef* and wrapped arrays for the varargs.

4.1.4 Pattern matching and closures

Pattern match inserts calls to *unapply*, *empty* or *get*. Closures need some additional information. Details of these two can be found in [8].

4.1.5 Future Scala language work

Future work will consist mainly of maintaining the inserted calls up to date. We should also explore mechanisms to register them directly in the phases were they are added or detect them automatically.

4.2 Calls on java code

Whenever we call a method for which we don't have a *MethodSummary* such as a method defined in java we can't we have to use other ways to know which methods could be called. It is important to know which methods can be called from java code as the following example makes evident.

```
// In Scala source
class Foo {
  def foo(): String = toString() + "!"
  def toString(): String = "Foo"
  }
System.out.println(new Foo)
```

```
// In Java java.io.PrintStream source
public void println(Object x) { ... x.toString() ... }
```

It is obvious that the *toString* of *Foo* will be called from the *println*, unfortunately this information is nowhere in the Scala source. Hence the normal approach would bailout when reaching the *println*, probably dead code eliminating *Foo.toString* and failing at runtime.

4.2.1 Current solution

The solution is simple, we need to add to the *println* all potential calls that it could do. This implies all methods that can be called on any of the arguments (including the *this*) then it can call any method on the return types of those method (and on the return types of those methods, and so on until a fixed point is reached). There would be two methods on *Foo*, 33 methods on *PrintStream* and 65 methods on *String* (see red nodes on the right of figure 1). These are an approximation of the actual calls but it ensures that all reachable code at runtime is reachable in the call graph, most likely with calls that will never be reached such as *Foo.foo*.

We take advantage of the knowledge that some of the sources were compiled before. Such information can reduce the number of potential calls, as for example when *PrintStream* was compiled it had no idea of the existence of *Foo* and hence could never call *Foo.foo*. We can also remove method that we know that can not be defined in our Scala code such as final methods or methods in final classes.

4.2.2 Future work on Java call graph

Another approach to reduce the number of calls is constructing the calls based on method names found in the class file constant table. This approach should reduce considerably the possible calls on the parameters and the calls on the return type of the parameters. This will make the graph more precise and probably faster to compute as it would reduce the calls that will need to be expanded. A step further would be to reconstruct the list names of methods called by each method. This would mainly help to reduce the number of calls on *this* as all the names of these methods will be present in the constant table.

4.3 Call graph visualization

Originally the call graph could be exported as a graphviz *.dot* file. This visual representation was key while debugging some bugs in the call graph, specially see and understand how some calls are reached (or are not reached). Unfortunately this format does not scale with the number of nodes that we generate, calling one function on the standard library *Predef* is enough to make it unintelligible. A new interactive graph visualizer that can scale to larger and more detailed graphs was implemented to solve the previous shortcomings. This one is exported to a stand alone HTML file that uses *vis.js* [6] to layout and manipulate the graph. The following snippet of code produces the graph in figure 1.

```
object Test {
  def main(args: Array[String]): Unit = Bar().test()
  }
class Bar {
  def test() = {
    class Foo { def bar: Int = 42 }
    val foo = new Foo
    System.out.println(foo.bar)
  }
  }
```

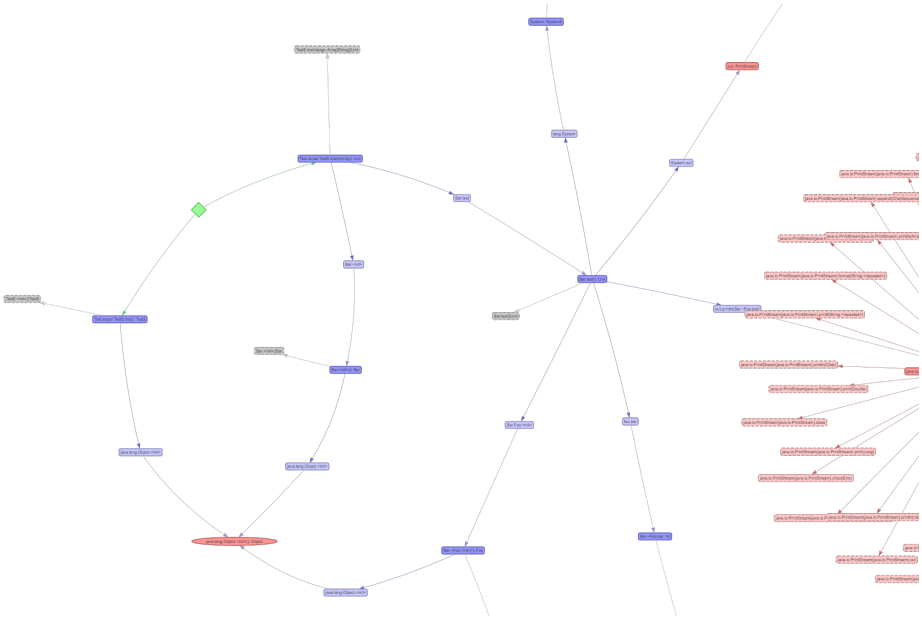

Figure 1: Call graph visualization of the code snippet above. The green node is the entry point to the *main* method and it's module, blue are calls to Scala code and red are calls to Java code.

5 Conclusion

Currently the call graph is up to date with *dotty/master* and supports the language features found in the standard library. There are still a couple bugs that need to be fixed to support some of the most complex code patterns in the standard library. The call graph also was extended to support calls to methods defined in bytecode, a requirement to create a complete call graph (compiled or not with the standard library). Future work will consist mainly of maintaining inserted calls up to date, fixing the remaining bugs and improving reducing the possible calls from bytecode methods.

References

[1] Call graph pull request for dotty/master. https://github.com/lampepfl/dotty/pull/1840.

[2] Dotty-linker project. https://github.com/dotty-linker/dotty.

[3] Dotty project. https://github.com/lampepfl/dotty.

[4] Java language and jvm specifications. https://docs.oracle.com/javase/specs/.

[5] Scala programming language. http://scala-lang.org/.

[6] vis.js. http://visjs.org/.

[7] *Call graphs for languages with parametric polymorphism.* OOPSLA, 2016.

[8] Romain Beguet. Call-graph-based optimizations in scala. Technical report, EPFL, 2016.

[9] M. Odersky. The scala language specification version 2.9. 2014. http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[10] D. Petrashko. Dotty linker: Making your scala applications smaller and faster. https://d-d.me/talks/scaladays2015/#/, 2015.