Remaining issues for Spark on 2.12

Lukas Rytz and Stavros Kontopoulos

As far as we know, there are two major remaining issues blocking a Spark release for Scala 2.12

- Overloaded methods can become ambiguous when used from Java 8
- Spark-14540 add support for the new lambda encoding to the closure cleaner

Note: any new issues will be reported here as well since this is WIP.

Proposed Action/Implementation Plan

Issue 1: Overloaded Methods

Do Nothing. Don't try to fix this as this is only a problem for Java users who would want to use 2.11 binaries. In that case they can cast to `MapFunction` to be able to utilize lambdas. In Spark 3.0.0 the API should be simplified so that this issue is removed.

Issue 2: Closure Cleaner

- Keep the exact same logic as today: if the class name contains `$anonfun`, run the existing closure cleaner.
    - This could be improved (some issues mentioned later in the doc), but that's out of scope.
- In case we have a lambda that does not have a "$anonfun" then use `getClass.isSynthetic && getClass.getInterfaces.contains(SerializableClass)`, then call writeReplace reflectively (example) and see if that returns a SerializedLambda. From there, find the body method, load the bytecode with asm, analyze it.
- Identify new LMF closures to make sure there are no `return` statements in their bodies.

Unit/integration tests will have to be adapted according to the above.

Issue 3: Overloading resolution for functions fails with Unit adaptation

This problem is described here. It was reported recently here. There is an easy workaround, we should proceed with, until the Scala bug is fixed. That is to change:

def addTaskCompletionListener(f: (TaskContext) => Unit): TaskContext
to
def addTaskCompletionListener[U](f: (TaskContext) => U): TaskContext

which is a binary compatible change, and it should be source compatible too.

Issue 4: Avoid reflection for accessing `globalFuture` in SparkIloop

Details of the issue are given here. Solution: TBD.

Alternatively in order to avoid touching globalFuture we should consider fixing the initial problem defined here and solved via a hack here. The initial problem came up due to the removal of loadFIles() in ILoop in Scala 2.11.12 (Preferred).

Detailed description of the major issues comes next along with alternative approaches.

Issue 1: Overloaded Methods

A document from March 2016 by Josh Rosen outlines the problem, but we'll repeat the relevant parts here, since the document is slightly out of date.

The Spark API has overloaded methods of the following form:

```
class Dataset[T] {
  def map[U : Encoder](func: T => U): Dataset[U] = ...
  def map[U](func: MapFunction[T, U], encoder: Encoder[U]): Dataset[U] = ...
}
```

For Scala clients, this works fine for 2.12 (the original document mentions a problem also for Scala clients, but this was resolved in the final 2.12.0 version).

However there is an issue for Java clients using this API. In the Java bytecode, the API has these overloaded methods:

```
  <U> Dataset<U> map(scala.Function1<T,U> f, Encoder<U> e) { ... }
  <U> Dataset<U> map(MapFunction<T, U> f, Encoder<U> e) { ... }
```

If the API is compiled with Scala 2.11, Java 8 code can use a lambda: `d.map(x -> y, enc)`. The Java compiler will select the second overload, because the first one takes a `Function1`, which is not a SAM type, so it is not applicable to a lambda.

If the API is compiled with Scala 2.12, `scala.Function1` is now a SAM type. This makes the invocation in Java code ambiguous.

Proposed solutions:

- Do nothing
    - This has no negative impact for Scala users, neither on 2.11 nor 2.12.
    - Java 8 users can continue to use the spark_2.11 artifacts in the same way.
    - Java 8 users that switch to the spark_2.12 artifacts cannot use lambda syntax for invoking these overloaded methods (or they have to insert an explicit cast).
- Keep the overloads as they are, but mark the first one with the `ACC_SYNTHETIC` flag in bytecode.
    - This would not change anything for Scala clients, the Scala compiler does not read these flags for Scala-generated class files.
    - The Java compiler then ignores the first overload, the `d.map(x -> y, encoder)` expression compiles (and selects the second overload).
    - This solution needs to be tested more widely to see if other Java compilers (eclipse, IntelliJ) have the same behavior.
    - There are multiple options for implementing this:
        - A change to the Scala compiler, which we can do in 2.12.7 (prototype here). The Scala 2.11 compiler does not need to change.
        - A Scala compiler plugin could do the same for existing compiler releases
        - A post-processing step in the build could modify the class files using ASM
- Remove the first overload

    - This cannot be done on Scala 2.11 for two reasons: it's not binary compatible, and Scala 2.11 clients could no longer use lambda syntax.
    - The idea of providing an implicit conversion on Scala 2.11 from scala.Function1 to MapFunction has a severe drawback: Scala code would always need explicit parameter types in the lambda expressions, i.e., `(x: Int) => y` instead of `x => y`
    - Removing the overload only in the 2.12 version (by using separate source files, or the enableIf macro) would make the 2.11 and 2.12 APIs slightly different. It seems this is undesirable.
- [EDIT: the following idea is not an option due to binary compatibility constraints. According to Imran Rashid, Spark 2.4.x needs to be binary compatible with Spark 2.3.x (when compiled on the same Scala version). Changing a method into a macro breaks binary compatibility.]
  Another new idea (by Jason Zaugg) is to turn the first overload into a macro
    - This would hide the first overload from Java clients
    - However, the original document mentions that the solution "Should not rely on unstable / esoteric Scala features, such as macros"
    - Furthermore, the change would not be binary compatible. I don't know if this would be a problem: Does Spark 2.4.0_2.11 (compiled with Scala 2.11) need to be binary compatible with Spark 2.3.1_2.11?

The API can be simplified once Scala 2.11 support is dropped.

Issue 2: Closure Cleaner

Introduction

Closure cleaner in Spark allows the cleaning of closures passed at different calls at the Spark API level. For example this is how closures are used at the API level:

```
def filter(f: T => Boolean): RDD[T] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[T, T](
    this,
    (context, pid, iter) => iter.filter(cleanF),
    preservesPartitioning = true)
}
```

The cleaning val `cleanF = sc.clean(f)` is done using this method:

```
private[spark] def clean[F <: AnyRef](f: F, checkSerializable: Boolean =
true): F = {
    ClosureCleaner.clean(f, checkSerializable)
    f
  }
```

`ClosureCleaner.clean()` is the entry point for the closure cleaner code and it is called at some places in Spark code base but they are not that many and all cases are like the example above.

Scala 2.12 generates closures differently, similar to Java 8 lambdas. The current closure cleaner is not designed for such closures. This is one of the remaining issues that blocks support of Scala 2.12 in Spark.

Closure Cleaner in 2.11

In Scala 2.11, the closure cleaner does the following

- Detect closures by class name (`cls.getName.contains("$anonfun$")`), return if the test fails
    - This is actually broken for classes defined within closures, see [1]
- Find any `throw NonLocalReturnControl` in the `apply` method of the closure. Throw a `ReturnStatementInClosureException` to the user if any.
- Find all outer classes (by the `$outer` field) of the closure.
    - Stop at (but include) the first object which is not a closure (by the name test above)
- Find all superclasses of these outer classes (these are the classes that declare fields that the closure might access)
- Find all closure classes to which the current closure is passed as argument (`getInnerClosureClasses`)
    - Done by bytecode traversal: find all constructor calls that take the current closure class as first parameter type
    - The goal is to find only closure classes, but in fact the method also returns other classes to which the closure is passed, see [2]
- Find all fields (of this or outer classes) that are actually accessed, either by the closure itself or by a closure nested within it
- For the outer closure chain, starting at the outside
    - clone the object (the outermost is only cloned if it's a closure) using reflection
    - set accessed fields in the clone, leave others `null`
    - clean the cloned closure transitively
        - I don't see which case this covers. The closure is part of the `$outer` chain of the original closure being cleaned. All of its unused fields are already cleaned, as far as I can tell.
        - Is it about finding `return` statements?

- If the starting closure doesn't actually need our enclosing object, then just null it out
    - Not sure if this can ever happen. The closure should not capture the outer object if it doesn't need it (?)

[1] The following generates a class named `A$$anonfun$1$B$1`

```
class A {
  val f = () => { class B; new B }
}
```

[2] For the following closure, `getInnerClosureClasses` includes `A$$anonfun$6$K$1`

```
class A {
  val f = () => {
    val x = 0
    class K { val y = x }
    val k = new K
    () => k.y
  }
}
```

Scala 2.12 Lambda Generation

Scala 2.12 generates closures using the same infrastructure as Java 8 lambdas: the LambdaMetaFactory (LMF). Instead of generating a class file for each lambda

- the body (code) of the lambda is emitted as a method in the enclosing class
- an InvokeDynamic instruction is emitted in bytecode, which passes all necessary information to the LambdaMetaFactory to create the closure class and instance at runtime

Basic example, with a nested function:

```
class A {
 val x = 0
 def f1 = () => () => x
}
```

This generates (pseudo-code, simplified):

```
class A {
  Function0 f1() {
    InvokeDynamic(
      this, // captured value
```

```
      LMF,
      Function0, // function type
      ()Object, // abstract method
      A.$anonfun$f1$1, // body method
      N // flags, e.g. for the closure to (implement) Serializable
  }

  static Function0 $anonfun$f1$1(A a) {
    InvokeDynamic(
      a, // captured value
      LMF,
      JFunction0$mcI$sp, // function type
      ()I, // abstract method
      A.$anonfun$f1$2, // body method
      N // flags
  }

  static int $anonfun$f1$2(A a) { a.x }
}
```

Note that lambda body methods are [always static](#) (in Scala 2.12, 2.13). If a closure captures the enclosing instance, the static body method has an additional `$this` parameter. The Java 8 compiler handles this differently, it generates instance methods if the lambda accesses state from the enclosing object.

Examples with local definitions:

```
class A {
  val x = 0
  val f2 = () => {
    def y = 1 // lifted, becomes method `A.y$1`. becomes a static method, as
              // it doesn't use the `A` instance.
    () => y   // does not capture anything. calls the static method `A.y$1`.
  }
  val f3 = () => {
    val y = 1 // local variable in the body method `$anonfun$f3$1`, passed
              // as argument to LMF when the nested closure is created
    () => y   // body method `$anonfun$f3$2` takes an int parameter
  }
  def m = {
    val z = x
    () => z   // `z` becomes a field in the closure, both in 2.11 and 2.12
  }
}
```

Nested classes

```
class A {
  val x = 0

  class B {
    val y = 1
    val f4 = () => x // captures the B instance b, accesses b.$outer.x
    val f5 = () => y // captures the B instance for b.x, does not use b.$outer
  }
}
```

Note that in the last example, the captured `B` instance is not cloned and cleaned. The closure cleaner only clones and cleans enclosing closures, not arbitrary objects.

Observations

Random, potentially relevant information: closures generated with LMF do not have a `serialVersionUID` field. Scala 2.11 closure classes always define this field (with value 0L).

As far as I can tell, all of the "cleaning" logic of the closure cleaner is unnecessary for 2.12. There is actually a simple argument: closure objects created by LambdaMetafactory never have an `$outer` pointer.

Closures are all lifted to the enclosing class as methods, arguments and captured values become parameters. Local definitions get lifted to the enclosing class, local variables in a function are passed by parameter to nested functions.

I also looked at the existing tests (ClosureCleanerSuite, ClosureCleanerSuite2, ProactiveClosureSerializationSuite). I think none of these examples would need cleaning in 2.12, but this should be checked again in more detail.

The functionality of finding non-local returns needs to be re-implemented for 2.12.

Identifying Closures

Identifying closure objects (which need to be checked for non-local returns) is actually not straightforward. Using the class name as in 2.11 (which is actually incorrect, as shown in earlier in this document) is not possible, as the classes are generated by the JDK (class names may differ depending on the JVM implementation).

A question on StackOverflow is answered by Brian Goetz that there is no way to do this, by design.

As a workaround, Jason Zaugg suggested to serialize the closure object (after checking if it conforms to `Serializable`). This returns an instance of `java.lang.invoke.SerializedLambda`, which in turn points to the enclosing class and the method implementing the lambda body. In a different StackOverflow post, Brian Goetz calls this a "really bad idea". Compared to the current closure cleaner, it doesn't look like a big hack though.

If we only need to check for non-local returns, there might be a different way. Instead of checking the closure object, maybe we can get a handle on the enclosing class, find all its closure body methods (with `$anonfun` in their name), and check those. I don't know if there is an entry point for such a check.

Different state capture in 2.12

When defining classes within lambdas, the state captured by a lambda may change. The reason is that in 2.11, a class defined in a lambda body is nested within the lambda class. Its `$outer` points to the lambda object:

```
class C {
  def f = () => {
    class A extends Serializable
    serialize(new A)
  }
}
```

In 2.12, the class A is lifted into the enclosing class C, therefore its `$outer` pointer now points to the `C` instance. In simple cases, the `$outer` pointer is set to `null` if it is not used. This is explained in more detail in
https://www.scala-lang.org/news/2.12.0/#sam-conversion-precedes-implicits.

Non-LMF closures in 2.12

There are a number of cases where Scala 2.12 does not use LambdaMetafactory but generates closure classes at compile-time (like 2.11):

- If the lambda is defined in a super constructor argument [1] - this is an implementation restriction in Scala
- If the SAM type does not compile down to a pure Java interface [2], i.e.,
    - traits with fields or concrete methods
    - abstract classes
- If the user sets the `-Ydelambdafy:inline` compiler flag

The same closure cleaner as in 2.11 should continue to work on such closures.

[1]

```
class A(f: Int => Int)
// generates a class file `B$$anonfun$$lessinit$greater$1`
class B extends A(x => x)
```

[2]

```
trait T { def f(x: Int): Int } // pure Java interface
trait U { def f(x: Int): Int; def m = 0 } // Not a pure Java interface
abstract class K { def f(x: Int): Int }
```