Syntax and optimizations for implicit use cases

Material adapted from http://jorgeortiz85.github.com/ImplicitClassSIP.xhtml which is

Abstract

A minimal language construct is proposed that allows for more concise use of the common “type trait” and “extension method” patterns in Scala. This construct permits the compiler to eliminate object creation so long as the transformation preserves the semantics of the original program. In addition, an annotation is proposed for the standard library that verifies such an optimization was performed.

Motivation

The popular extension method patter, sometimes called the Pimp My Library pattern is used in Scala to extend pre-existing classes with new methods, fields, and interfaces.
There is also another common ‘extension’ use case known as type traits or type classes (see scala.math.Numeric). Type classes offer an alternative to pure inheritance hierarchies that is very similar to the extension method pattern.
The main drawback to both of these techniques is that they suffer the creation of an extra object at every invocation to gain the convenient syntax. This makes these useful patterns unsuitable for use in performance-critical code. In these situations it is common to remove use of the pattern and resort to using an object with static helper methods. In many cases, this is a simple mechanical transformation that could be performed by an optimizing compiler.

This proposal outlines inline semantic changes

Description

The proposal is actually composed of two separate but correlated proposals.

- Improving the flexibility of the `@inline` annotation optimisations in the compiler to include classes.
- An `implicit class` syntax

Improved inlining

The compiler shall be adapted to be able to inline the instantiation and usage of methods of a class in an expression if an instance of the class does not escape the expression. This analysis occurs after all method inline expansion occurs. That is, if a method returns an instance of the class, but is marked for inlining and the class itself is marked for inlining, then the entire instantation can still be inlined if the class does not escape the expanded expression.

For example, given the following class and object pairing:

```
@inline
class Foo(x: Int) {
  def plus(y: Int) = x + y
  override def toString = "Foo("+x+")"
}
object Foo {
  @inline final def apply(x: Int): Foo = new Foo(x)
}
```

The following expressions could be inlined:

```
new Foo(1) plus 2
new Foo(1) toString
new Foo(3) plus (_: Int)
Foo(1) plus 2
Foo(1).toString
Foo(3) plus (_: Int)
```

And the following expressions would not be inlined:

```
new Foo(1)
(x: Int) => new Foo(x)
Foo(_:Int)
```

In the example :

```
foo(1).toString
```

the inliner would first expand the Foo.apply method call, making the expression:

```
new Foo(1).toString
```

Then during the inline class pass, the instance of Foo created is seen to not escape the the expression, and the entire call is inlined into equivalent code for:

```
"Foo(" + 1 + ")"
```

The suggested implementation is for the compiler to generate static methods for the Foo class and delegate to them on inlining, unless they are themselves inlined. This expression would actually be:

```
Foo.methods$.toString(1)
```

The goal of inling class is to remove the object instantiation, not necessarily remove the method call. However, an inlined class my also have its methods marked inline. In this case, an optional third pass of the inliner could attempt to remove any method calls after class inlining has been acheived. This allows expressions involving inlined classes to inline all code relating to a method, as opposed to delegating to static methods for the class.

So given this class:

```
@inline
class Foo(x: Int) {
  @inline def plus(y: Int) = x + y
}
```

and the expression:

```
new Foo(1) plus 2
```

The compiler would eventually inline this into:

```
1 + 2
```

Most likely, this third pass is uneeded for targetting the JVM and other backends. It is noted in the proposal in case the need does arrive.

In addition to the new inlining optimisations, the @inline annotations will be modified to accept an error behavior in its constructor of “WARN”, “FAIL” or “SILENT” with the default being “WARN”. If a class or method marked @inline is unable to be inlined by the compiler, the compiler will take the following actions:

- If the error behavior is WARN, than a warning is issued at the point where inlining could not occur. Compilation proceeds.
- If the error behavior is FAIL, then an error is issued at the point where inlining could not occur. Compilation is halted.
- If the error behavior is SILENT, then no action is taken, and the class is treated normally.

Implicit class syntax

The implicit keyword will now be allowed as an annotation on classes. This will have the following effect.

implicit classes

An *implicit class* must have a primary constructor with exactly one argument in its first parameter list. It may also include an additional implicit parameter list. An implicit class must be defined in a scope where method definitions are allowed (not at the top level). An implicit class is desugared into a class and implicit method pairing, where the implciit method mimics the constructor of the class.
The generated implicit method will have the same name as the implicit class. This allows importing the implicit conversion using the name of the class, as one expects from other implicit definitions.
For example, a definition of the form:

```
implicit class RichInt(n: Int) extends Ordered[Int] {
  def min(m: Int): Int = if (n <= m) n else m
  ...
}
```

will be transformed by the compiler as follows:

```
class RichInt(n: Int) extends Ordered[Int] {
  def min(m: Int): Int = if (n <= m) n else m
  ...
}
implicit final def RichInt(n: Int): RichInt = new RichInt(n)
```

Like case classes, implicit classes are still allowed to have companion objects. The implicit apply method will be added onto the existing companion class. If a companion object exists *and* defines an apply method, then TBD (Some kind of warning?).

Annotations on implicit classes default to attaching to the generated class *and* the method. For example,

```
@inline(WARN)
implicit class Foo(n: Int)
```

will desugar into:

```
@inline(WARN) implicit def Foo(n: Int): Foo = new Foo(n)
@inline(WARN) class Foo(n:Int)
```

The annotation.target annotations will be expanded to include a “genClass” and “method” annotation. This can be used to target annotations at just the generated class or the generated method of an implicit class. For example:

```
@(inline @genClass) implicit class Foo(n: Int)
```

will desugar into

```
implicit def Foo(n: Int): Foo = new Foo(n)
@inline class Foo(n: Int)
```

Specification

No changes are required to Scala's syntax specification, as the relevant production rules already allow for implicit classes.
LocalModifier ::= ‘implicit’
BlockStat ::= {LocalModifier} TmplDef
TmplDef ::= [‘case’] ‘class’ ClassDef
The language specification (SLS 7.1) would be modified to allow the use of the implicit modifier for classes. A new section on Implicit Classes would describe the behavior of the construct.

Consequences

Since this proposal only outlines additions and optional features to the language, it should not break existing code. However, the complexity of escape analysis and inlining could significantly impact compilation time.

inlined classes

The details of opimtizations are left to the compiler implementer, but some suggestions are envisioned to help achieve a common optimization strategy across the features in this proposal. Inlined classes could be implemented as follows:

```
@inline final class Bar(x: Int, y: String) {
  def foo(z: Double) = y + x + “-” + z
}
```

would compile as follows:

```
final class Bar(x: Int, y: String) {
  def foo(z: Double) = Bar.methods$.foo(x,y,z)
}
object Bar {
  <synthetic> object methods$ {
    <synthetic> final def foo($t1: Int, $t2: String, $p1: Double) =
        $t2 + $t1 + "-" + $p1
  }
}
```

And the expression:

```
new Bar(5, "Time-").foo(1.0)
```

would compile to:

```
Bar.methods$.foo(5, "Time-", 1.0)
```

Allowing the JVM inliner to continue inlining the foo method. Since this optimisations are at the compiler's disgresison, arbitrary constraints may be made on classes that can participate in inlining. The above suggestion of embedding 'static' methods for inlined classes rather than directly inlining is meant to strike a balance between generated code size and efficiency. It is the hope tha tthe JVM inliner will inline these static methods where it makes sense, but profiling alternatives will lead to a much better decision.

Requirements for inlined classes

As a possible first implementation of inlined classes, the following requirements will be in place:

- classes may only have other @inline annotated classes for parents
- classes may only define methods, the exception being constructor arguments.