Type Specialization in Scala 2.8

Iulian Dragos

May 6, 2010

1 Introduction

Scala uses type erasure to compile away parametric polymorphism (generics), which induces a performance cost that can be non-negligible. Essentially, type parameters are erased and replaced by their upper bound, defaulting to `Object`. It follows that code involving primitive types will have to be adapted to work on objects, by adding appropriate boxing and unboxing operations. Programs that manipulate boxed primitive values may suffer a slow-down of ten times compared to hand-specialized code. This in turn leads to programmers avoiding generic collections, and writing hand-written specialized versions over and over.

Scala 2.8 adds *specialized* type parameters. A specialized type parameter instructs the compiler to generate a generic version and a number of specialized versions of the given definition, and use the most specific version whenever the static type information at a call site allows it.

A more detailed account of how specialization works can be found in [1].

2 Syntax

Any type parameter of a method or class definition can be annotated with `@specialized`. Optionally, the programmer may indicate for which primitive types should that type parameter be specialized.

```
class Vector[@specialized A] {
  def apply(i: Int): A = //..
  def map[@specialized(Int, Boolean) B](f: A => B) =
    //..
}
```

The `Vector` above will specialize `A` for all primitive types, while method map will get a specialized version only for integers and booleans.

3 Implementation

Specialization in Scala is performed at *definition-site*, and is *eager*: the compiler derives specialized definitions for all combinations of primitive types, or a subset defined by the user. The following types are primitive types in Scala: Unit, Boolean, Byte, Short, Char, Int, Long, Float, Double. Specialization is performed at the definition site in order to allow separate compilation. If specialization was performed only at the point where a generic class is instantiated

to a primitive type, it may be the case that the original class definition is not compiled in the same run (this is the usual case, since we expect specialization to play an important role in libraries) and the compiler cannot derive specialized implementations for methods that might benefit from it.

A class definition may generate a set of specialized classes and a generic (erased) class. Each specialized class is derived from the original definition using a specific combination of types and extends the generic class. The key point of specialization in Scala is that programs contain both specialized and erased class definitions. For them to coexist, it is necessary that specialized instances can be used instead of their generic counterpart.

Consider the following example:

```
class RefCell[@specialized(Int) T] {
  private var value: T = _

  def get: T = x
  def put(x: T) =
    value = x
}
```

Scalac generates an additional class extending `RefCell[Int]`, which holds specialized versions of RefCell's members. It also overrides all generic member definitions to delegate to their specialized definitions. This code path involves boxing and is equivalent to using erasure, and is necessary to ensure correct behavior when a specialized instance is used by code that uses the generic interface. When generic code is used in a context where more type information is available, and there exists a specialized version, the compiler rewrites method calls and class instantiations to their specialized versions. This code path is guaranteed to avoid boxing.

```
class RefCell$mcI$sp extends RefCell/*[Int]*/ {
  protected var value$mcI$sp: Int = _;

  protected override def value: AnyRef =
    value$mcI$ // boxing happens here

  protected override def value_=(x$1: Int): Unit =
    value$mcI$sp = x$1

  override def get: AnyRef =
    get$mcI$sp; // boxing happens here
  override def get$mcI$sp: Int = value$mcI$sp;

  override def put(x: AnyRef): Unit =
    put$mcI$sp(x); // boxing happens here
  override def put$mcI$sp(x: Int): Unit = value$mcI$sp = x
}
```

The class above specializes `RefCell` for type `Int`. It has a specialized integer field for holding the current value and overrides the accessors of the inherited field (`value`) to use it instead. In addition to the field, methods that mention the specialized type parameter are also overridden. Their implementation is rewritten by replacing references to `T` by `Int` and using a more specific field or method whenever possible.

Assume a client of this class that uses it to handle integers:

```
object Test extends Application {
  val ref = new RefCell[Int]

  ref.put(10)
  println(ref.get)
}
```

The Scala compiler will replace the instantiation of `RefCell[Int]` with the right specialized version, and calls to `put` and `get` won't perform boxing.

3.1 Member Specialization

Not all members of a class are specialized. Currently, specialized variants of a member m are created only when m's type contains at least one specialized naked type parameter, or an array of a naked specialized type parameter. For example:

```
abstract class Foo[@specialized T, U] {
  // the following members are specialized
  def foo1(x: T): U
  def foo2(x: Int): Array[T]
  val a: Array[T]

  // the following members are not specialized
  def bar1(x: U): Unit
  def bar2(x: List[T]): U
  val b: List[T]
}
```

4 Status

The current implementation in Scala trunk uses specialization by default. Specialization can be turned off by passing `-no-specialization` on the command line.

4.1 Standard Library Specialization

The following classes are specialized:

- `FunctionN` traits up to two parameters. Parameter types are specialized on Int, Long, Float and Double. The result type parameter is additionally specialized on Unit and Boolean.
- `AbstractFunctionN` is specialized on the same types as `FunctionN`.
- Tuples up to two parameters are specialized on Int, Long and Double.
- Method `Range.foreach` is specialized on Unit. This allows fast for-loops on integers, in the form

  ```
  for (i <- 1 until 100) // ..i..
  ```

References

[1] Dragos, I., and Odersky, M. Compiling generics through user-directed type specialization. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (New York, NY, USA, 2009), ACM, pp. 42–47.