Improving the Performance of Scala Collections with Miniboxing

EPFL Technical Report

Aymeric Genêt  Vlad Ureche  Martin Odersky
EPFL, Switzerland
{first.last}@epfl.ch

Using generics, Scala collections can be used to store different types of data in a type-safe manner. Unfortunately, due to the erasure transformation, the performance of generics is degraded when storing primitive types, such as integers and floating point numbers. Miniboxing [6] is a novel translation for generics that restores primitive type performance. Naturally, a good choice would be to use miniboxing to translate Scala collections. In this paper we explore the patterns used to implement the Scala collections, describe how they are transformed by miniboxing and finally compare the performance of the two transformations on a mockup of the Scala collection library. The benchmarks show our prototype implementation\(^1\) can speed up collection operations by 45% without any need for programmer intervention.

**Keywords** Scala, generics, specialization, miniboxing, primitive types

1. Introduction

Scala collections allow storing data in an abstract and type-safe manner. This is done using generics, which allow treating types as parameters of classes and methods. Using generics, it is possible to abstract over the type of the data in a collection, such as, for example, creating a linked list of integers. Safety is then guaranteed by the type system, which can statically prove that all elements of the collection are actually integers. This increases programmer productivity and improves the quality of both programs and libraries.

Generics are currently translated to low level bytecode using the technique of erasure [2]. This entails that all type parameters are replaced by their lower bound, which is usually `Object`. This is a convenient translation, since all generic values are uniformly represented as a references. However, this makes it impossible to instantiate the type parameters by primitive types, such as integers or floating point numbers, which are not references but values. Instead, an object representation of the primitive type must be used. This is done by wrapping primitive values into objects, in a process called boxing. The opposite process, which extracts the primitive value from an object, is called unboxing. Boxing and unboxing primitive values degrades program performance, inflates the heap memory requirements and triggers extra garbage collection cycles.

Miniboxing [6] is an alternative translation for generics, which avoids boxing and unboxing, thus improving the performance for primitive types. This is done by creating additional versions of the generic methods and classes, specifically adapted to accept primitive values as arguments and return primitive types. Instead of creating an additional version for each primitive type, which would be wasteful in terms of bytecode size, miniboxing creates a single version which can encode all primitive types. We call these additional versions of methods and classes specialized variants. Therefore, in the case of miniboxing, for a single type parameter, there will be two variants of a method or class: one using the (reference-based) erasure translation, which is used for objects, and a specialized variant, for primitive types.

Scala collections expose a simple and high-level interface. This allows programmers to effortlessly transform collections by mapping over their elements, filtering them or splitting collections based on custom criteria. All these features make heavy use of generics and are thus affected by slowdowns when used with primitive types. This makes Scala collections unsuitable for numeric processing applications, such as machine learning or bioinformatics.

This naturally leads to the idea of translating Scala collections using the miniboxing transformation in order to reap the benefits of the convenient high-level interface while offering good performance for numeric applications. Yet this is not an easy task: collections are implemented using multiple layers of functionality and use complex patterns in order to reduce code duplication and gain flexibility.

In this paper, we set out to use the miniboxing transformation on a mock-up of the Scala collections, which includes all the relevant patterns used in the real collections. In this context, we make the following contributions:

- we explain the patterns that implement Scala collections;
- we show how the miniboxing transforms each pattern;
- we benchmark our mock-up of the Scala collections using both the erasure and miniboxing transformations.

The benchmarks show promising results: the miniboxing plugin [1] can speed up collection operations by 45% without any need for user intervention.

The paper first describes the miniboxing transformation, implemented as a plugin for the Scala compiler. Then, it describes the code patterns used in implementing Scala collec-

\(^1\)http://scala-miniboxing.org
tions and their transformation with the miniboxing plugin, especially: (1) the class hierarchy, (2) the closures, (3) the Builder pattern, and (4) the Numeric pattern. Finally, the last section presents our mockup collections and the benchmark results.

2. Miniboxing

Miniboxing [1, 6] is a compilation scheme that improves the performance of generics in the Scala programming language. The miniboxing transformation is activated by annotating type parameters with @miniboxed, for both classes and methods:

```scala
class C[miniboxed T](val t: T) {
  def foo(): T = t
}
```

In order to understand the transformation behind this annotation, let’s look at the following example. Assume we want to write a generic method that will likely be used with primitive type arguments:

```scala
def bar(t: T): T = (new C[T](t)).foo()
```

Since the type parameter T of the bar method is marked as @miniboxed, the method will have two versions: the erasure-based, slow version of the method and its specialized variant for primitive types, which encodes all primitive types (from booleans to double-precision floating point numbers) as long integers:

```scala
def bar(t: Object): Object = 
  new C_L(t).foo(...) // erasure-based version
def bar_J(T_Tag: byte, t: long): long = 
  new C_J(....).foo_J(....) // specialized variant
```

Every time the programmer calls bar with a primitive type parameter, the compiler rewrites the code to call the optimized version bar_J. However, when the method is called with an object type parameter, such as String, the erasure-based method will be called:

```scala
bar[String]("x") // bar is used
bar[Int](3)     // bar_J is used
```

Now let us look at the miniboxed class C. The first step will be to transform the class into an interface and create the specialized variants for accessors and methods:

```scala
trait C {
  def t: Object // erasure-based getter for t
  def t_J(...): long // specialized getter for t
  def foo(): Object
  def foo_J(...): long
}
```

Now, C will have two implementations, C_L and C_J:

```scala
class C_J(T_Tag: byte, t: long) extends C {
  def t: Object = minibox2box(T_Tag, t_J(...))
  def t_J(...): long = t
  def foo(): Object = minibox2box(...)
  def foo_J(...): long = t
}
```

For brevity, we omit the implementation of C_L, which is similar, only that t is a reference and T_Tag disappears.

In the user code, an instantiation of C with a primitive type parameter is rewritten to an instantiation of C_J. An instantiation with a reference parameter leads to an instantiation of C_L. The method calls are also rewired in order to match the expected types: If one uses the method foo in the optimized C_J, it’s actually the foo_J method that must be called:

```scala
val c_s = new C[String]("x") // class C_L is used
val c_i = new C[Int](3)   // class C_J is used
println(c_s.foo())     // foo_J is used
println(c_i.foo())     // foo_J is used
```

Finally, the last type of specialization occurs when transforming superclasses and mixins:

```scala
class D extends C[Int](2) {
  // class D extends class C_J
  class E[miniboxed T](t: T) extends C[T] {
    // trait E extends trait C
    // class E_J extends class C_J
    // class E_L extends class C_L
}
```

The process of class specialization brings two important advantages: Firstly, since class fields are specialized, the performance of accessing fields will improve, because the program now deals with a direct value access instead of a reference-based access. This is done by representing fields as long integers, instead of objects. When necessary, the conversions between the long integer representation and the object representation are added by the transformation (minibox2box and box2minibox). Secondly, the memory footprint of the class will be reduced, since storing data in its primitive format requires less memory than creating a new object and storing a reference to it.

3. Scala Collections

In this paper we show how the miniboxing transformation enables improved collections, which expose the same high-level interface without sacrificing performance. The next section presents the common patterns that enable the high-level interface in the Scala collections [5], and how miniboxing can be applied in order to improve performance.

3.1 Inheritance and Mixins

Inheritance and mixins group the common behavior of different collections. This reduces code duplication and gives rise to a convenient collection hierarchy, where each level of the inheritance makes more assumptions about the architecture than the previous level. For example, the path to a linked list goes through Traversable, Iterable, Seq, LinearSeq and finally List.

This nesting and splitting of functionality makes it necessary to have deep miniboxing: Adding the @miniboxed annotation to a collection type parameter will not be enough to fully transform it, as most of its functionality will be inherited from parent traits. Instead, what needs to be done is to deeply visit all the parent traits and mark their arguments as @miniboxed:
In Scala, it is common to use functions to manipulate collections. For example, in order to extract the positive numbers in a List of integers, we can use the `filter` method along with the following function:

```scala
List(4,-2,1).filter(x => x > 0)
```

However, since the Java Virtual Machine doesn’t support functions (at least not until Java 7), Scala needs to provide a special translation for them:

```scala
List(4,-2,1).filter{
  class $anon extends Function1[Int, Boolean] {
    def apply(x: Int): Boolean = x > 0
  }
  new $anon()
}
```

The `Function1` trait is provided by the standard library and can’t be overridden with a miniboxed version. Hence, in order to specialize functions, we need to provide our own function traits, which are miniboxed and perform the desugaring by hand.

This is done by creating a custom `MyFunc1` trait that receives two type parameters, `T` and `R`, which signal the argument and return of our function, i.e `(T => R)`. This trait exposes an abstract `apply` function that will contain the actual code of the function. Miniboxing is triggered by annotating both of the type parameters with `@miniboxed`:

```scala
trait MyFunc1[-T, +R] extends Object {
  def apply(t: T): R
}
```

Then, just like methods, four different abstract traits that extend the previous interface will be created.

Now, in order to express the previous function, we can write:

```scala
new MyFunc1[Int, Boolean] { def apply(x: Int): Boolean = x > 0 }
```

And the miniboxing transformation will translate this to:

```scala
new MyFunc1_JJ[Int, Boolean] { ... }
```

Now, any invocation of this function will actually invoke `apply_JJ`, thus completely avoiding boxing primitive types, such as `int` and `boolean`.

### 3.3 Builder Pattern

The Builder pattern is the key component necessary for collection comprehensions: It greatly reduces code duplication, since all the collection comprehensions reduce to creating a new collection with either transformed or filtered elements. It also brings flexibility, as shown by the following example:

```scala
scala> val map = Map(1 -> 2, 2 -> 3)
map: immutable.Map[Int,Int] = ...

scala> map.map((case (x, y) => (y, x)))
res1: immutable.Map[Int,Int] = ...

scala> map.map((case (x, y) => x))
res2: immutable.Iterable[Int] = ...
```

To achieve this, the `map` function will rely on a `Builder` generated from the `CanBuildFrom` parameter, where `Repr` is the current collection and `That` is the resulting collection:

```scala
def map[R, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That = {
  val b = bf(repr) // the builder
  for { x <- this } b += f(x)
  b.result
}
```

The Builder pattern also shows how type constructor polymorphism can play an essential role in factoring out boilerplate code without losing type safety [4].

### 3.4 Numeric Pattern

Defining a generic type for a class can sometimes lead to inconvenient situations if one expects to have generic mathematical operations. Since the common ancestor for numeric types, `Any`, does not contain mathematical operations, the operations on generic types are quite limited.
The Numeric pattern solves this issue by allowing mathematical operations in a generic context. This is done by creating a generic Numeric trait that provides mathematical operations for a certain type. By example, one could define a way to add two numerical values, by providing such a definition to the trait:

```scala
trait Numeric[T] {
  def plus(x: T, y: T): T
  ...
}
```

We can extend the trait into different concrete implementations, that provide operations for each primitive type individually. This pattern also works for non-primitive number representations such as BigInteger. For instance, the code for Numeric[Int] would be:

```scala
implicit object NumInt extends Numeric[Int] {
  def plus(x: Int, y: Int): Int = x + y
  ...
}
```

Now, every time we want to use a type parameter as a numeric type, we enforce that the Numeric version of the type exists, so we can call the mathematical operations on them. Here is a complete example of a two-dimensional vector class:

```scala
class Vec2D[T : Numeric](val x: T, val y: T) {
  def n = implicitly[Numeric[T]]
  new Vec2D[T] {
    n.plus(this.x, that.x),
    n.plus(this.y, that.y)
  }
  ...
}
```

Since the Numeric implementations are likely to use primitive type parameters, boxing and unboxing would frequently occur. This is where the miniboxing specialization steps in. With a simple @miniboxed annotation on the type parameter of the Numeric class, a concrete extension would override an optimized version for primitive types. The classes that use the Numeric objects should also have a @miniboxed annotation. This would avoid every occurrence of boxing and unboxing, and greatly enhance the performance.

### 4. Benchmarks

To benchmark performance, we implemented a mockup of the Scala collections library including Traversable, Iterable, Iterator and a linked list class with its builders. We also implemented other necessary parts of the Scala library, including Function1, Tuple2 and the Numeric pattern. To assess the speedup of the miniboxing plugin used on the collections mockup, we implemented a common numerical application: the least squares method. This method computes the parameters of a linear equation that best describes a given set of points. Since the benchmark deals with numbers, type erasure introduces boxing and unboxing operations. The method should therefore run faster with the miniboxing plugin. The benchmark used is the following:

```scala
// list of (x,y) coordinates
val xy = xs.zip(ys)

// function (x, y) => x * y
val fxy = new Function1[Tuple2[Double,Double], Double] {
  def apply(t: Tuple2[Double, Double]): Double = t._1 * t._2
}

// function x => x * x
val fxx = new Function1[Double, Double] {
  def apply(x: Double): Double = x * x
}
```

If we run one version of the above benchmark with our mock-up collection, one with the plugin activated (Miniboxed) and another without (Generic), for different amount of points, we get the following results:

<table>
<thead>
<tr>
<th>Amount of points</th>
<th>Miniboxed [ms]</th>
<th>Generic [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>160</td>
<td>279</td>
</tr>
<tr>
<td>2000000</td>
<td>328</td>
<td>557</td>
</tr>
<tr>
<td>3000000</td>
<td>487</td>
<td>831</td>
</tr>
</tbody>
</table>

The results show 40-50% speedups when miniboxed collections are used, without any intervention on the programmer’s side. It is also worth noting that scala specialization [3], the current solution used in the Scala compiler, was not able to successfully compile the example.

### References

2. The benchmarking code is identical for the two transformations, only the library contains miniboxed annotations.