# Miniboxing and the MbArray API

Semester Project

Romain Beguet

École polytechnique fédérale de Lausanne, Switzerland
{first.last}@epfl.ch

## 1. Introduction

In modern programming languages, genericity allows abstracting over types, enabling programmers to develop algorithms and data structures regardless of the data being handled. This tremendeously improves productivity and code reuse. Although generics offer a uniform programming experience accross different types, the actual data comes in different sizes and shapes. For example, we can equally instantiate generic container for 1 bit boolean values, 32 bit integer values, or even object types passed by reference. To resolve the tension between the uniform types and non uniform nature of data, compilers take two different approaches:

**Homogeneous compilation** imposes a common shape for all incoming data. Its most common implementation is called erasure, and is typically the scheme employed by languages that target the Java Virtual Machine, such as Java, Scala or Kotlin, which use an object representation even for primitive types. The downside of this approach is that it affects runtime performance: Primitive types such as integers or booleans need to be encoded as objects when entering a generic context, in a process called boxing, which negatively impacts performance.

**Heterogeneous compilation** duplicates and adapts the generic code for each primitive type. This allows it to efficiently handle data of different sizes and shapes. In Scala, this is implemented through the specialization transformation.

While specialization is great for performance, the amount of low-level code it generates makes it impractical. The reason is that each generic class is duplicated 10 times, corresponding to the 9 primitive types in Scala plus an erased version that is used for objects. Furthermore, for classes with $n$ type parameters, specialization generates the cartesian product of their specializations, producing as many as $10^n$ duplicates. This is where the miniboxing approach comes in, reducing the amount of duplication down to $3^n$.

Miniboxing is a middle ground between heterogeneous and homogeneous approaches, encoding several primitive types into a larger one and thus reducing the duplication factor, without paying the price of boxing. In most benchmarks, miniboxing matches the performance of specilization, while generating significantly less low-level code.

In the context of generic programming, one of the problems is implementing bulk storage, exposed in most languages through arrays. Since imposing a homogeneous translation to bulk storage would be terribly inefficient, the current approach is to use specialized arrays, even in generic code. However, in erasure-based homogeneous compilation, all generic type information is erased from the low-level program, such that there is no way for it to know which array to instantiate at runtime. This makes it necessary to have special objects which carry the type information explicitly, commonly known as reified types. In the case of Scala, these objects are called `ClassTag`s:

```
scala> def foo[T] = new Array[T](10)
<console>:7: error: cannot find class tag for element
   type T
       def foo[T] = new Array[T](10)
                    ^
```

In practice, carrying class tags for generic code is expensive and needs to be done transitively through the entire code base.

`MbArray` is an indexed sequence container that matches the performance of raw arrays when used in miniboxed contexts. Also, unlike arrays, `MbArray` creation does not require the presence of a class tag, which makes it more versatile. The `MbArray` data structure was created based on some underlying assumptions, which directly impacts its performance and usability.

In this context, the contributions of this semester project are:

- Explicitly stating the underlying assumptions of the `MbArray` bulk storage container.
- Developping benchmarks that challenge the underlying assumptions of the `MbArray` data structure, and identifying which of them are valid and which need to be revisited.
- Revisiting one of the underlying assumptions of `MbArray`s, proposing an improved approach, and implementing it in practice.
- Validating the improved design using multiple benchmarks.

## 2. The Miniboxing Transformation

This section describes the miniboxing transformation, showing why it is necessary, how it improves the program performance and how it has an opportunistic nature.

### 2.1 Generics in Scala

Generics are crucial to productivity. They allow programmers to design algorithms and data structures that operate identically regardless of the data used, fostering code reuse. For example, a `Vector[T]` can be instantiated for any type, whether for numbers, strings or CPU threads. However, on the low level, data comes in different shapes and sizes: from 1-bit booleans to 64-bit long integers, floating point numbers, characters, value classes [1, 3, 6] and objects. In a `Vector[Int]`, getters and setters receive integer values of 32-bits while in `Vector[Double]` they receive 64-bit double-precision floating-point numbers.

The current compilation scheme for generics is called erasure, and is the simplest compilation scheme possible for generics. Erasure requires all data, regardless of its type, to be passed in by reference, pointing to a heap object. Let us take a simple example, a generic `identity` method written in Scala:

```scala
def identity[T](t: T): T = t
val five = identity(5)
```

When compiled, the source code corresponding to the low-level bytecode for the method is:

```scala
def identity(t: Object) = Object
```

As the name suggests, the type parameter `T` was "erased" from the method, leaving it to accept and return `Object`s. The problem with this approach is that values of primitive types, such as integers, need to be transformed into heap objects when passed to generic code, so they are compatible with `Object`, in a process called boxing. The process goes two-ways: the return of generic methods needs to be unboxed back to a primitive type:

```scala
val five = identity(Integer.valueOf(5)).intValue()
```

Boxing primitive types requires heap allocation and garbage collection, both of which slow down program performance. Furthermore, when values are stored in generic classes, such as `Vector[T]`, they need to be stored in the boxed format, thus inflating the heap memory requirements and slowing down execution. In practice, generic methods can be as much as 10 times slower than their monomorphic (primitive) instantiations. This gave rise to a simple and effective idea: specialization.

### 2.2 Specialization

Specialization is a second approach used by the Scala compiler to translate generics. It is triggered by the `@specialized` annotation added to a type parameter:

```scala
def identity[@specialized T](t: T): T = t
val five = identity(5)
```

Based on the annotation, the specialization transformation creates several versions of the `identity` method:

```scala
def identity(t: Object): Object = t
def identity_I(t: int): int = t
def identity_C(t: char): char = t
// ... and another 7 versions of the method
```

Having multiple methods, also called specialized variants or simply specializations of the `identity` method, the compiler can optimize the call to `identity`:

```scala
val five: int = identity_I(5)
```

This transformation side-steps the need for a heap object allocation, improving the program performance. However, specialization is not without limitations. As we have seen, it creates 10 versions of the method for each type parameter. But what if a method has 2 type parameters? It creates 100 versions, and in general, for $N$ specialized type parameters, it creates $10^N$ specialized variants, the cartesian product covering all combinations. This prevents the Scala library from using specialization extensively, since common classes have between one and three type parameters. This led to the next development, the miniboxing transformation.

### 2.3 Miniboxing

Taking a low level perspective, we can observe the fact that all primitive types in the Scala programming language fit within 64 bits. This is the main idea that motivated the miniboxing transformation: instead of creating separate versions of the code for each primitive type alone, we can create a single one, which stores 64-bit encoded values, much like a tagged union [4]. But unlike a tagged union, miniboxing can use the statically typed nature of the language to avoid carrying tags with each value. Looking at the previous example:

```scala
def identity[@miniboxed T](t: T): T = t
val five = identity(5)
```

Since the type parameter is annotated with `@miniboxed`, the compiler[1] translates the code to:

```scala
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
val five: int= minibox2int(identity_M(int2minibox(5)))
```

Alert readers will notice the `minibox2int` and `int2minibox` transformations act exactly like the boxing coercions in the case of erased generics. This is true, the values are being coerced to the miniboxed representation, much like in the case of erasure, but on the Java Virtual Machine platform our benchmarks have shown that the miniboxing conversion cost is completely eliminated when compiling the code to native x86 assembly. Further benchmarking has shown that the code matches the performance of specialized code within a 10% slowdown due to coercions [7].

There is an elipsis in the definition of the `identity_M` method, which stands for what we call a type byte: a byte describing the type encoded in the long integer, allowing the operations such as `toString`, `hashCode` or `equals` to be executed correctly on encoded values, treating them as the original primitive (corresponding to `T`) rather than long integers. Although the transformation looks simple, there are many subtleties we have to take into account when transforming the code [2, 7, 8].

## 3. Arrays

Arrays provide efficient memory usage and constant random access time, thanks to the elements being stored in a contiguous fashion. This is also optimal from a low level perspective as it cooperates well with processor caches, thanks to the spatial locality of the elements. For these reasons, arrays are commonly used as the underlying storage in high level collections, and are the container of choice for implementing performance-critical algorithms.

Since elements of an array are stored contiguously in memory, accessing an element requires knowing its size: assuming an integer is encoded on 4 bytes, a compiler will know that the $n^{\text{th}}$ element of an integer array stored at the address $a$ can be accessed by de-referencing the address $a + 4 \times n$. Or, in the case of a higher-level virtual machine, such as the JVM, to avoid tying the implementation to architecture details, each primitive array, such as `int[]` or `double[]` has its own bytecodes to access and update elements, which de-reference the correct address for the given architecture.

Unfortunately, while using primitive arrays offers optimal performance and memory usage, it also makes it difficult to abstract over the type of elements in the array: different types of elements have different sizes so the compiler needs to generate different instructions for array access depending on its element type. In practice, abstracting over the elements in arrays is essential for writing generic data structures, so language developers proposed different approaches to enable abstraction. One example is the template-based way, used in C++ and .NET, where generics are instantiated to primitive types, resolving their size before execution. On the other hand, languages where generics are compiled with erasure, such as Scala, lose the information necessary to resolve the object size at runtime and need to employ more complex mechanisms.

[1] This occurs in the presence of the miniboxing plugin attached to the Scala compiler. In the rest of the paper we assume the miniboxing Scala complier plugin is active unless otherwise noted. For more information on adding the miniboxing plugin to the build please see `http://scala-miniboxing.org`.

A naive solution, commonly used in dynamic language interpreters which lack static information, is to convert all values stored in the array to object references, through boxing. This fixes the size of objects, since only references are stored in the array regardless of the primitive type. However, compact storage and locality are lost, since boxed primitives contain the additional object headers and can be allocated anywhere on the heap.

A much better approach transforms the Scala code that accesses or updates generic arrays. For example, the code:

```
def first[T](a: Array[T]): T = a(0)
```

Is transformed by the Scala compiler to:

```
def first(a: Object): Any =
  a match {
    case x: Array[Object] => x(0) // size = reference
    case x: Array[Int] => x(0)    // size = integer
    case x: Array[Double] => x(0) // size = double
    // and so on for all the primitive types of Scala
  }
```

With this transformation, which is similar to the accessor technique presented at the beginning of the section, accessing generic arrays becomes very involved: the program has to match over the possible primitive arrays, and only then it can access the element. This slows down program execution, but it also enforces a strong invariant: a Scala value of type `Array[Int]` is always represented as the JVM primitive integer array, `int[]`, and never as `Array[java.lang.Integer]`, which would require boxing the primitive values. This invariant guarantees that, outside the generic context, array access/update is efficient: an `Array[Int]` is always represented as the JVM primitive `int[]` array, which allows Scala to access and update elements without matching the array type. Therefore, from a language perspective, programmers only pay the extra overhead when using arrays with generics.

Another thing to notice is that the `Array[T]` type in the signature of `first` is transformed into `Object`: at the JVM level, primitive arrays such as `int[]`, `float[]` and `Object[]` have a single common parent class, which is `java.lang.Object`. This explains why there is a need to match the primitive array type: the `Array` class offered by Scala does not really exist, it's simply an abstraction over the primitive arrays offered by the JVM.

## 3.1 Instantiating Arrays

Although the array representation invariant looks like a good deal so far, it introduces an unexpected problem: how to instantiate generic arrays? Exactly like accessing or updating an array, instantiating one must abide by the array invariant. Yet, with the erasure transformation, all traces of a generic type parameter `T` are removed, transforming the code:

```
def newArray[T] = new Array[T](10)
```

into:

```
def newArray() = /* error: T is erased */
```

The `newArray` method should be able to produce any primitive array, based on the type argument of `newArray`:

```
val a1: Array[Long] /* = long[] */ = newArray[Long]
val a2: Array[Char] /* = char[] */ = newArray[Char]
```

**Reified types.** To address this issue, Scala offers the `ClassTag` mechanism, which can be used to provide runtime information about a type parameter. This allows `newArray` to produce the correct primitive type:

```
def newArray[T: ClassTag] = new Array[T](10)
```

is compiled to:

```
def newArray(tag: ClassTag) = tag.newArray(10)
```

Since the `ClassTag` object carries runtime information about the type `T`, it can instantiate the correct primitive array. Still, this approach is rarely used in practice because `ClassTag`s are expensive to synthesize, propagate and store. To illustrate this, let us consider the following code:

```
def foo[T] = bar[T]
def bar[T] = baz[T]
def baz[T] = new Vector[T](10)
```

If we want to create a generic `Array` instead of the `Vector`:

```
def baz[T] = new Array[T](10) // error: need ClassTag
```

But this requires `baz` to have a `ClassTag` passed in:

```
def baz[T: ClassTag] = new Array[T](10)
```

Therefore, to call method `baz`, the caller needs to pass in a `ClassTag` object. When the call is made from non-generic code (e.g. `baz[Int]`), the `ClassTag` is synthesized by the compiler based on the type argument. This bears the cost of synthesis. And there is another cost, of propagation: the `baz` method is also a caller of `bar`, but since `bar`'s type parameter `T` is erased, the compiler can't synthesize a `ClassTag` object. Instead, it must require its callers to provide it:

```
def bar[T: ClassTag] = baz[T]
```

Similarly, `foo` must require the `ClassTag` object to be passed from its callers, thus producing the following code:

```
def foo[T: ClassTag] = bar[T]
def bar[T: ClassTag] = baz[T]
def baz[T: ClassTag] = new Array[T](10)
```

This shows that generic code must be transitively modified to propagate `ClassTag` objects in order to allow instantiating arrays, explaining the cost of propagation. Finally, when dealing with objects, `ClassTag` objects have to be stored as fields in the object itself, adding to the program's memory footprint, explaining the storage cost.

**Breaking the invariant.** Another way to deal with this issue, without using `ClassTag`s, is to always instantiate `Array[AnyRef]` and force the compiler believe it is an `Array[T]`:

```
def baz[T] =
    new Array[AnyRef](10).asInstanceOf[Array[T]]
```

This approach forces boxing all elements, so it is known to affect performance. But, even worse, it breaks the array invariant: if the type parameter `T` is instantiated by `Int`, the returned array has type `Array[Int]` but its runtime type is `Object[]` instead of `int[]`.

Therefore this approach can only be used when the array is guaranteed to not escape to outside code. In practice, this approach is used to implement many of the generic collections in the Scala library.

**Specialized arrays.** When using Scala specialization, the methods are duplicated and the type parameters are statically known. This allows the array invariant to be used to efficiently access arrays:

```
def first[@specialized T](a: Array[T]): T = a(0)
```

This will yield multiple variants of `first`, the default – slow and unoptimized – one, as well as nine fast specializations:

```
def first(a: Object): Object =
  a match {
    ... // all ten cases
  }
def first_I(a: int[]): int = a(0)
def first_J(a: long[]): long = a(0)
// and 7 other specialized variants...
```

In this case, the generic (inefficient) version of the `first` method is still called in two cases:

- from erased generic code;
- when the type parameter of `first` is instantiated by a reference type;

Still, optimizing the specialized versions does not eliminate the need for `ClassTag` objects:

```
def newArray[@specialized T]: Array[T] =
  new Array[T](10)
```

Which is translated to:

```
def newArray: Object = // error: no ClassTag
def newArray_I: int[] = ...
def newArray_J: long[] = ...
// and 7 other specialized variants ...
```

Inside the generic version of `newArray`, `T` is erased, so there is no way to instantiate the array anymore. And, as we have seen before, we cannot assume that the generic `newArray` will always be called with a reference type parameter:

```
def foo[T] = newArray[T]
val arr: Array[Int] = foo[Int]
```

Therefore, the only correct solution is to require `newArray` to accept a `ClassTag` object, even before the specialized variants are created:

```
def newArray[@specialized T: ClassTag]: Array[T] =
  new Array[T](10)
```

Therefore, while specialization does reduce the overhead of acessing primitive arrays, it does not eliminate the need for `ClassTag` objects.

**Recap.** We have seen three ways to implement generic arrays:

- The `ClassTag` approach:
  - *Advantage:* Solves the array instantiation problem;
  - *Disadvantage:* Has to carry `ClassTag` objects;
- Breaking the invariant approach:
  - *Advantage:* Solves the array instantiation problem;
  - *Disadvantages:* Requires boxing primitives and arrays must not escape;
- The specialization approach:
  - *Advantage:* Optimizes array accesses;
  - *Disadvantage:* Does not solve the generic array instantiation problem.

## 4. Miniboxed arrays.

Would it be possible to combine the three approaches seen so far to produce an alternative `Array` which does not have any of the disadvantages? In this subsection we present how `MbArray` achieves this, keeping the main advantages of each approach and dropping their disadvantages.

`MbArray` is a generic class that wraps a Scala `Array` but is also able to guarantee it does not escape, thus being similar to the "Breaking the invariant" approach. This allows programmers to instantiate `MbArray` objects without the need for a `ClassTag` object. It may seem this path will lead to boxing values in the array, but this is not the case: thanks to the tight integration with miniboxing, the `MbArray` class can reflectively decide what array to instantiate: either an array of objects or a specialized variant containing miniboxed values. Therefore, the underlying array does not box the elements unless the type argument is erased at the instantiation site.

### 4.1 MbArray's Underlying Assumptions

The `MbArray` class is implemented with several design decisions in mind:

1. Instead of using `ClassTag`s, use the miniboxed status to decide which version of the array to create.
2. Wrap the array inside an object which doesn't allow it to escape, so that any inconsistent decision can't be observed from the outside.
3. Inside the array, use the miniboxed representation (`long` or `double`) instead of the unboxed representation, as this sidesteps the need for the tag-based dispatch and the primitive conversion in the access procedures.

The last point is explained in the next subsection.

### 4.2 MbArray Implementation

This subsection will give an insight into the implementation of `MbArray`s.

**Instantiation.** An `MbArray` object is instantiated either by cloning an already existing `MbArray` object through the `clone` method, or by creating an empty array of a given size this way:

```
val ary = MbArray.empty[Int](3)
```

At compile time, when the miniboxing transformation operates, every `MbArray` instantiation is rewired to one of the specialized constructors: `mbArray_empty_L`, `mbArray_empty_J` or `mbArray_empty_D`, which will instantiate the specialized variants of the `MbArray`:

- `MbArray_L` for object references;
- `MbArray_J` for integral types;
- `MbArray_D` for floating point types;

Of course, the instantiation will get rewired to an optimized constructor if and only if the instantiation is monomorphic or occurs in a miniboxed context. For example, the instantiation from the code snippet above will be transformed to:

```
val ary = mbArray_empty_J(3, MiniboxingConstants.INT)
```

On the other hand the following code cannot be rewired to a specialized constructor, since the instantiation occurs in a generic context that is not miniboxed:

```
def makeArray[T] = MbArray.empty[T](3) // will
   instantiate the generic MbArray where primitives
   are boxed
```

This can be solved by annotating the type parameter `T` with the `@miniboxed` annotation.

**Access.** Capitalizing on the tight integration with miniboxing, the array access procedures are transformed in a very similar fashion to optimistically access the underlying array directly, obtaining the miniboxed value right away. This is an opportunistic approach: there is a fast-path, when the storage type of the miniboxed class and the `MbArray` object coincide, or a slow path, when they don't.

But, thanks to the guidance from the performance advisories [2], the latter case only occurs with the programmer's consent, who knowingly allows the program to take the slow path.

With this approach, the following code:

```
def first(a: MbArray[Int]): Int = a(0)
```

Is translated to:

```
def first(a: MbArray): Int =
  minibox2int(mbArray_apply_J(a, 0))
```

Where the `mbArray_apply_J` method contains the optimistic assumption that the `MbArray` is going to be the specialized version of the `MbArray` class, containing an underlying array of long integers. Here is the Scala code equivalent to the real implementation of the `mbArray_apply_J` method (in Java):

```
def mbArray_apply_J[T](T_Type: Byte, a: MbArray[T],
    idx: Int): Long =
  a match {
    case x: MbArray_J => x.apply_J(idx)
    case x: MbArray_L => box2minibox(x.apply_L(idx))
  }
```

There are two possibilities:

- Going through the fast path does not require boxing but only a conversion from the miniboxed encoding to the unboxed integer: `x.apply_J(idx)` directly retrieves the miniboxed value from the underlying `Array[Long]`. This explains the third assumption presented in subsection 4.1: since the value is already in its miniboxed representation inside the storage, there is no need to match over the type of the underlying array and to convert the primitive value.
- On the other hand, the slow path accesses an array of objects and transforms the boxed value into a miniboxed one, paying the cost of handling the boxed value.

### 4.3 Conclusion

To conclude, the opportunistic nature of the `MbArray` coupled with performance advisories and miniboxing integration, allows it to have the following properties:

- Solves the array instantiation problem (no `ClassTag`s);
- Stores miniboxed values, which are more efficient than boxing;
- Accesses elements efficiently (if advisories are heeded).

## 5. Benchmarking MbArrays

In high-level languages, arrays are rarely used explicitly in algorithms, as they are usually provided with a limited API which only offers basic operations such as element retrieval and update. Instead, as it has already been stated in the last section, arrays are more often used as the underlying storage of more high-level collections, such as `ArrayBuffer`s and `Vector`s.

In order to measure performance improvements brought by `MbArray`s combined with the miniboxing transformation in realistic scenarios, I implemented the `ArrayBuffer` collection. It was implemented in two ways: using the common `Array`s with `ClassTag`s, and using `MbArray`s with miniboxing. Hence, the only differences *in the source code* between the two versions appear:

In the class declaration:

```
// The MbArray version
class ArrayBuffer[@miniboxed T](var _size: Int)
      extends ...

// The ClassTag version
class ArrayBuffer[T: ClassTag](var _size: Int)
      extends ...
```

And where arrays are instantiated:

```
// The MbArray version
      _array = MbArray.empty[T](_capacity)

// The ClassTag version
      _array = new Array[T](_capacity)
```

### 5.1 High-level Overview of the ArrayBuffer

Quoting the official Scala documentation, an `ArrayBuffer` is:

> *An implementation of the Buffer class using an array to represent the assembled sequence internally. Append, update and random access take constant time (amortized time). Prepends and removes are linear in the buffer size.*

On top of this, high-level features are provided with the implementation, such as `foreach`, `map` or `filter`. These methods are regularly used because they allow programmers to enhance their productivity significantly. For this reason, these methods became the actual subjects of the benchmarks.

### 5.2 ArrayBuffer implementation

In this subsection, I explain how the main features of the `ArrayBuffer` are implemented. Note that code snippets are taken from the miniboxed version.

**Dynamic size**. One of the properties of the `ArrayBuffer` data structure is that elements can be added at any time, thus requiring the `ArrayBuffer` to dynamically resize itself. For example:

```
val buf = new ArrayBuffer[Int](2)
buf(0) = 1
buf(1) = 2
buf.append(3) // buf now contains {1, 2, 3}
```

However, this feature is not proposed by default by the underlying array: neither `Array` nor `MbArray` can be resized. Thus, a naive way to implement `append` is to re-instantiate the underlying array with a size $n + 1$ each time an element is appended, where $n$ is the previous size of the array, and to then copy the content of the old array into the new one. While it is simple to implement, it has major performance issues in case several `append` calls are done sequentially to a decently large `ArrayBuffer`.

Instead, the growth strategy commonly employed is to allocate an array of two times the size of the previous one, which ultimately produces less garbage while considerably reducing the number of full array copies that need to be done. Here is how the `append` method was implemented:

```
def append(elem: T) = {
  if (_size >= _capacity) {
    val old = _array
    _capacity *= 2
    _array = MbArray.empty[T](_capacity)
    ArrayBufferUtils.copyAll(old, _array);
  }

  _array(_size) = elem

  _size += 1
}
```

[2] Compiler warnings triggered by the miniboxing plugin when it detects suboptimal code

In the code snippet above, `_size` represents the actual number of elements that the `ArrayBuffer` contains, and `_capacity` the maximum number of elements that the `ArrayBuffer` can contain before having to resize itself.

**Iterable**. Iterating over the entire collection is an operation that is recurrent in algorithms. These algorithms often want to make abstraction of the collection type that is used and only want to express their need to traverse it. `Iterable` is a common interface which is implemented by every collection that supports complete traversal. It requires its implementations to provide access to an `Iterator`, which will itself be used to iterate through the collection. To achieve this, the `ArrayBuffer` implements the `Iterable[T]` interface presented below:

```
trait Iterable[@miniboxed T] {
  def iterator: Iterator[T]

  def foreach(f: (T) => Unit) = {
    val it = iterator
    while (it.hasNext) {
      f(it.next)
    }
  }
}

trait Iterator[@miniboxed T] {
  def next: T
  def hasNext: Boolean
}
```

**Buildable**. Occasionnally, algorithms also want to express their need to build a new collection based on an existing one. In the Scala Standard Library, this feature is exposed through the `CanBuildFrom` mechanism [5]. Here, I decided to simplify (but not too much) the design by proposing an interface which only allows building the same type of collection. That is, a `Collection[T]` can only build `Collection[U]`. Thanks to Scala's ability to abstract over type constructor, the interface could be implemented in a straightforward manner:

```
trait Buildable[@miniboxed T, Container[_]] extends
    Iterable[T] {
  def builder[@miniboxed U]: Builder[U, Container]

  def map[@miniboxed U](f: T => U) = {
    val bd = builder[U]
    val it = iterator
    while (it.hasNext) {
      bd.append(f(it.next))
    }
    bd.finalise
  }

  def filter(f: T => Boolean) = {
    val bd = builder[T]
    val it = iterator
    while (it.hasNext) {
      val elem = it.next
      if (f(elem)) {
        bd.append(elem)
      }
    }
    bd.finalise
  }
}

trait Builder[@miniboxed T, Container[_]] {
  def append(x: T): Unit
  def finalise: Container[T]
}
```

The `Builder` implementation for the `ArrayBuffer` is done the following way:

| Size | ClassTag | MbArray |
|---|---|---|
| 300000 | 5.41 ms | 5.92 ms |
| 600000 | 9.92 ms | 10.71 ms |
| 900000 | 14.44 ms | 13.63 ms |
| 1200000 | 18.85 ms | 22.49 ms |
| 1500000 | 22.88 ms | 24.30 ms |

**Table 1.** The initial ScalaMeter benchmark's outputs

```
class ArrayBufferBuilder[@miniboxed T] extends
    Builder[T, ArrayBuffer] {

  var innerBuf: ArrayBuffer[T] = new ArrayBuffer[T](0)

  override def append(elem: T) = innerBuf.append(elem)
  override def finalise = innerBuf
}
```

## 5.3 Benchmarking procedure

In order to benchmark run time performance of `MbArray`s, I created two projects – one for each `ArrayBuffer` version – which use the Scalameter library. The original benchmark implementation is the following:

```
import org.scalameter.api._

import mbvector._

object MbVectorBenchmark extends
    PerformanceTest.Quickbenchmark {
  val sizes = Gen.range("size")(300000, 1500000,
      300000)

  override def executor = new
      org.scalameter.execution.LocalExecutor(
    Warmer.Default(),
    Aggregator.average,
    measurer)

  val bufs = for {
    size <- sizes
  } yield new ArrayBuffer[Int](size)

  performance of "ArrayBuffer" in {
    measure method "map" in {
      using(bufs) setUp {
        b =>
          b.map(_ + 1)
          b.map(_ + 2)
      } in {
        b => b.map(_ + 1)
      }
    }
  }
}
```

The program will bench 5 different `ArrayBuffer` configurations corresponding to 5 different sizes of `ArrayBuffer`s from 300'000 to 1'500'000 elements. For each configuration, `ScalaMeter` performs multiple runs of the code being benchmarked and yields – in our case – the average of each run. It is worth noting that before each run, the `ScalaMeter` benchmarking procedure runs the code multiple times in order to make sure the JVM has as little noise as possible and that the JIT has compiled the methods.

## 5.4 The Numbers

Unfortunately, the numbers produced by running the benchmark above did not look as good as we could have (rightfully) expected, as it can be seen in figure 1. Surprisingly, the `MbArray` version showed to be approximately 10% slower than the `ClassTag` version, which is completely contradictory with what has been stated earlier!

```
object Benchmark {

  def vecSize = 10000000
  def opCount = 20

  def makeBuffer(size: Int, fill: Int => Int) = {
    val vec = new ArrayBuffer[Int](size)
    vec.map(fill)
  }

  def time(opName: String, count: Int, init: =>
     ArrayBuffer[Int], operation: ArrayBuffer[Int] =>
     Unit) = {
    var i = 1
    var total = 0L

    println("ArrayBuffer. " + opName + " : ")

    while (i <= count) {
      var vec = init

      println("--- next iteration")
      val start = System.currentTimeMillis()
      operation(vec)
      val end = System.currentTimeMillis()
      println("\t" + i + ". : " + (end - start) + "ms");

      vec = null
      System.gc()

      total += end - start
      i += 1
    }

    println("Total : " + total + "ms. Average " +
       (total.toDouble / count) + ".\n")
  }

  def main(args: Array[String]) = {

    time("map", opCount, {
      makeBuffer(vecSize, i => 1)
    }, {
      _.map { _ * 2 }
    })

  }
}
```

**Figure 1.** Handmade implementation of the initial benchmark. The benchmark is done on an ArrayBuffer of 10M elements

## 6. Challenging the Assumptions

In this section, I describe the steps that I followed in order to debug the slowdown observed in the last section, that is, to identify its cause.

### 6.1 Analyzing of the Numbers

First of all, I tried to tweak the parameters of the benchmark to try deducing the cause of the slowdowns. As one could expect, modifying basic items such as the size of the `ArrayBuffer`s did not improve the numbers. Ultimately, I had to dig deeper into what could cause the problem, and to this end rewrote the benchmarking process by hand so that it would be easier to debug and to experiment with the JVM flags. Usually, benchmarking is done through a dedicated framework, such as ScalaMeter, but in this particular case I needed to isolate the benchmark and be able to run it directly. The handmade benchmark implementation can be found in figure 1. Thinking back about the original problem, one of the cause for the slowdown could have been that hot methods would not get inlined properly. Indeed, the miniboxing transformation generally produces methods that are larger than their generic counterparts, which can prevent inlining. This can be easily detected with a debug build of the HotSpot virtual machine and the `-XX:+PrintCompilation` flag. Unfortunately it did not pay off, as everything was inlined as expected. Another direction was to obtain a breakdown of the time spent computing and collecting garbage. Indeed, running the benchmark once more with the `-XX:+PrintGC` option yielded a more interesting result: in the version using `MbArray`s a GC cycle gets triggered *during* the benchmark iteration, whereas none happen for the `ClassTag` version:

```
   12. : 255ms
[GC 328332K->641K(2009792K), 0.1071670 secs]
[Full GC 641K->641K(2009792K), 0.1344740 secs]
--- next iteration
   13. : 249ms
[GC 328333K->641K(2009792K), 0.1078760 secs]
[Full GC 641K->641K(2009792K), 0.1345190 secs]
--- next iteration
   14. : 250ms
[GC 328325K->641K(2009792K), 0.1071550 secs]
[Full GC 641K->641K(2009792K), 0.1355260 secs]
```

(a) ClassTag version

```
   12. : 375ms
[GC 328502K->197268K(2067968K), 0.8788130 secs]
[Full GC 197268K->660K(2067968K), 0.2077410 secs]
--- next iteration
[GC 524954K->197396K(2066048K), 0.1963980 secs]
   13. : 374ms
[GC 328512K->197268K(2071680K), 0.8838360 secs]
[Full GC 197268K->660K(2071680K), 0.2077530 secs]
--- next iteration
[GC 524953K->197364K(2069888K), 0.1962730 secs]
   14. : 374ms
[GC 328466K->197268K(2075008K), 0.9097610 secs]
[Full GC 197268K->660K(2075008K), 0.2010080 secs]
```

(b) Miniboxed version

**Figure 2.** The initial benchmark's outputs for ArrayBuffers of 10'000'000 elements

It is important to note that in the printouts above, the GC cycles that are printed after the end of an iteration and the start of a new one are triggered manually by a `System.gc()` and are not counted in the time duration of an iteration.

We can thus infer the following:

- The `ClassTag` version spends no time collecting garbage during the iteration, but collects 330MB in 242ms at the end of it.
- The `MbArray` version spends up to 196ms collecting 330MB of garbage during the iteration, and collects 330 additional Megabytes at the end of the iteration in 1090ms.

Note that the benchmarks were executed enough times to trigger:

- JIT compilation with the server compiler (C2).
- Full inlining of the code being benchmarked, which enabled escape analysis and therefore eliminates boxing even in generic code.

Moreover, through profiling, we made sure no boxed values (`java.lang.Integer`) were created. Thus, the only differences in the low-level code are the array accessors: for the `ClassTag` version, the accessors are `ScalaRunTime.array_apply` and `ScalaRunTime.array_update`, while for the `MbArray` version, they are `mbArray_apply_J` presented earlier as well as `mbArray_update_J`.

We can therefore derive that the cause of the slowdown is due to an overly high amount of garbage being produced by the version using `MbArray`s.

### 6.2 Problem with the Current Design

In order to understand the cause for the high amount of garbage being produced, it is necessary recall how `MbArray`s are currently transformed by the miniboxing plugin.

**Memorandum.** As explained in subsection 4.2, when an `MbArray` is instantiated in a context where its type argument is known to be a primitive type, the miniboxing plugin transforms at compile time the instantiation of the generic `MbArray` into one of its specialized version: `MbArray_J` for an integral type, or `MbArray_D` for a floating point type. At first, it seemed like a good idea to have only these two specialized versions since internally, the miniboxing plugin only deals with three representations: `Object`, `long` and `double`, and therefore, storing the values as their miniboxed representation avoids the array type dispatch and coercion costs that would occur in the access procedures if they were stored as their true representation. This corresponds to the third assumption presented in subsection 4.1.

**Back-of-the-enveloppe Calculation.** However, the GC printouts from figure 2 show that using the miniboxed representations internally causes significantly more memory to be used when storing integers. In order to confirm the nature of the additionnal memory usage, we did the following calculation:

First, the total number of elements stored for each benchmark iteration can be computed as follows:

1. The initial array is 10M elements large.
2. In the `init` function, we build another `ArrayBuffer` from the initial one through mapping, which produces intermediate `ArrayBuffer`s of 1, 2, . . . 8M and finally 16M elements due to the growth strategy employed, summing up to a total of 32M elements.
3. Inside the benchmark code, we are mapping one more time, thus adding another 32M elements `ArrayBuffer`
4. In total, we are storing 74M elements per iteration.

We can now separate the calculation for the two versions:

- For the `ClassTag` version, the elements stored are integers. Therefore, since an integer is 4 bytes long, we are holding 74M $\times$ 4B $=$ 296MB. Since we can safely consider a 30MB overhead, we attain the same number that we find in practice in figure 2(a): 330MB.
- For the `MbArray` version however, elements stored are not integers but longs. Since a long is 8 bytes long, we are holding 74M $\times$ 8B $=$ 592MB. Moreover, the miniboxing plugin introduces an extra overhead on top of the 30MB due to having miniboxed values of 64bits and type tags being passed around a lot, but also due to miniboxed functions taking twice as much memory. We can thus safely assume a 60MB overhead, giving us the amount of memory that we find in practice in figure 2(b): 660MB.

Hence we can observe that for integers, the `MbArary` version of the `ArrayBuffer` uses twice as much memory as the `ClassTag` version. Even worse, for booleans, which only need one bit, storing values in the long representation causes the heap foopring to increase 64 times. In turn, this leads to more GC cycles, which slow down the program execution.

## 7. Redesigning MbArrays

After having exposed the flow in the current `MbArray` design that slows down its performance, I present the approach I took in order to address it.

### 7.1 The Goal

The goal is to keep the mechanisms in `MbArray`s that are working, but to decrease the memory needed for that. To this end, the design change I opted for was adding more specializations to the `MbArray`s: one for each primitive type that exists in Scala. In other words, an `MbArray[Boolean]` would get transformed to `MbArray_B` which uses a `boolean[]` internally instead of the prevous `long[]`, and so on equivalently for every of the 10 primitive types that exist in Scala. Hopefully, this would result in less heap memory wasted and therefore a higher GC throughput.

| Size | ClassTag | Old MbArray | New MbArray |
|---|---|---|---|
| 300000 | 5.41 ms | 5.92 ms | 4.84 |
| 600000 | 9.92 ms | 10.71 ms | 9.38 |
| 900000 | 14.44 ms | 13.63 ms | 12.02 |
| 1200000 | 18.85 ms | 22.49 ms | 18.18 |
| 1500000 | 22.88 ms | 24.30 ms | 21.26 |

**Table 2.** The initial ScalaMeter benchmark's outputs, including the new MbArray version's results.

### 7.2 Design transformation

Adding the 7 other specializations was fairly straightforward: All that had to be done was duplicating one of the two existing specialization for every new variants and changing the internal array type `Array[Long]` to the ones adapted for the different specialization. Then, the code that handled `MbArray` instantiations, namely `MbArray_empty_J`, `MbArray_empty_D`, `MbArray_clone_J` and `MbArray_clone_D`, had to be modified in order to take into accounts the new specializations. For example:

```
public static <T> MbArray<T> mbArray_empty_J(int size,
   byte T_Tag) {
 return new MbArray_J<T>(T_Tag, size);
}
```

Was transformed to:

```
public static <T> MbArray<T> mbArray_empty_J(int size,
   byte T_Tag) {
 switch(T_Tag) {
 case MiniboxConstants.LONG:
   return new MbArray_J<T>(size);
 case MiniboxConstants.INT:
   return new MbArray_I<T>(size);
 case MiniboxConstants.SHORT:
   return new MbArray_S<T>(size);
 case MiniboxConstants.CHAR:
   return new MbArray_C<T>(size);
 case MiniboxConstants.BYTE:
   return new MbArray_B<T>(size);
 case MiniboxConstants.BOOLEAN:
   return new MbArray_Z<T>(size);
 case MiniboxConstants.UNIT:
   return new MbArray_V<T>(size);
 default:
   return new MbArray_L<T>(size);
 }
}
```

Similarly, the `apply` and `update` methods had to be changed. For example:

```
public static <T> long mbArray_apply_J(MbArray<T>
   mbArray, int index, byte T_Tag) {
 if (mbArray instanceof MbArray_J<?>)
   return ((MbArray_J<?>)mbArray).apply_J(index);
 else
   return MiniboxConversionsLong.box2minibox_tt(
       mbArray.apply(index), T_Tag);
}
```

Became:

```
public static <T> long mbArray_apply_J(MbArray<T>
  mbArray, int index, byte T_Tag) {
  if (mbArray instanceof MbArray_J<?>)
    return ((MbArray_J<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_I<?>)
    return ((MbArray_I<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_S<?>)
    return ((MbArray_S<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_C<?>)
    return ((MbArray_C<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_B<?>)
    return ((MbArray_B<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_Z<?>)
    return ((MbArray_Z<?>)mbArray).apply_J(index);
  else if (mbArray instanceof MbArray_V<?>)
    return ((MbArray_V<?>)mbArray).apply_J(index);
  else
    return MiniboxConversionsLong.<T>box2minibox_tt(
      mbArray.apply(index), T_Tag);
}
```

## 7.3 Re-benchmarking

Following the transformation, we re-benchmarked out example programs using the new `MbArray`.

**Initial benchmark.** Table 2 shows that the new version performs approximately at the same level as the `ClassTag` version on the benchmark. Moreover, figure 3 (which shows the results of the handmade version of the initial benchmark) shows how the new `MbArray` version, just as the `ClassTag` version does not spend any time collecting garbage *during* an iteration, from which follows a 200ms runtime performance gain over the old `MbArray` version. In average, the new version is 15% faster than the old version in the ScalaMeter benchmark and 215% on the handmade benchmark.

Unfortunately, as one can notice, even the new `MbArray` version does not seem to be a lot faster than the `ClassTag` version on that initial benchmark, only reaching a small 5% speedup.

**Mapping to Floats.** However, most of the benchmark *do* show better numbers for the new `MbArray` version. For example, another benchmark was developped, which shows that the version using new `MbArray`s can be approximately 250% faster than the version using `ClassTag`s and 20% faster than the version using the old `MbArray`s. This can be seen in table 3. The source code of this benchmark is almost identical to the initial one, the only difference being that instead of:

```
  using(bufs) setUp {
    b =>
      b.map(_ + 1)
      b.map(_ + 2)
  } in {
    b => b.map(_ + 1)
  }
```

It is doing :

```
  using(bufs) setUp {
    b =>
      b.map(_ + 1)
      b.map(_ + 2)
  } in {
    b => b.map(_ + 1).map(_ + 2.5f).map(_ + 3)
  }
```

Thus building `ArrayBuffer`s containing `Float`s. Also, figure 4, which corresponds to the handmade version of this new benchmark, shows how the version using new `MbArray`s spends approximately 650% less time collecting garbage compared to the old version.

**Mapping to Longs and Doubles.** There is another case which needs to be benchmarked, corresponding to the best case scenario for old `MbArray`s: mapping to `Long`s and `Double`s. The reason it is the best scenario is that it is the one in which no extra memory is wasted by the old `MbArray` implementation. Here is the benchmark code:

```
  using(bufs) setUp {
    b =>
      b.map(_ + 1L)
      b.map(_ + 2L)
  } in {
    b => b.map(_ + 1L).map(_ + 2.5).map(_ + 1.5)
  }
```

Results for this benchmark can be seen in table 4, which show that we did not lose any run time performance by changing the design, even in the case where it is the most advantageous for the old version.

```
   12. : 255ms
[GC 328332K->641K(2009792K), 0.1071670 secs]
[Full GC 641K->641K(2009792K), 0.1344740 secs]
--- next iteration
   13. : 249ms
[GC 328333K->641K(2009792K), 0.1078760 secs]
[Full GC 641K->641K(2009792K), 0.1345190 secs]
--- next iteration
   14. : 250ms
[GC 328325K->641K(2009792K), 0.1071550 secs]
[Full GC 641K->641K(2009792K), 0.1355260 secs]
```

(a) ClassTag version

```
   12. : 375ms
[GC 328502K->197268K(2067968K), 0.8788130 secs]
[Full GC 197268K->660K(2067968K), 0.2077410 secs]
--- next iteration
[GC 524954K->197396K(2066048K), 0.1963980 secs]
   13. : 374ms
[GC 328512K->197268K(2071680K), 0.8838360 secs]
[Full GC 197268K->660K(2071680K), 0.2077530 secs]
--- next iteration
[GC 524953K->197364K(2069888K), 0.1962730 secs]
   14. : 374ms
[GC 328466K->197268K(2075008K), 0.9097610 secs]
[Full GC 197268K->660K(2075008K), 0.2010080 secs]
```

(b) Old Miniboxed version

```
   12. : 176ms
[GC 328350K->659K(2009792K), 0.0973560 secs]
[Full GC 659K->659K(2009792K), 0.1256140 secs]
--- next iteration
   13. : 171ms
[GC 328351K->659K(2009792K), 0.0933540 secs]
[Full GC 659K->659K(2009792K), 0.1280840 secs]
--- next iteration
   14. : 169ms
[GC 328343K->659K(2009792K), 0.0974840 secs]
[Full GC 659K->659K(2009792K), 0.1259070 secs]
```

(c) New Miniboxed version

**Figure 3.** The initial benchmark's outputs for ArrayBuffers of 10'000'000 elements, including the new MbArray version's results.

| Size | ClassTag | Old MbArray | New MbArray |
|---|---|---|---|
| 300000 | 22.68 ms | 11.65 ms | 9.47 ms |
| 600000 | 45.77 ms | 23.28 ms | 18.23 ms |
| 900000 | 57.04 ms | 29.02 ms | 24.13 ms |
| 1200000 | 93.44 ms | 44.66 ms | 35.91 ms |
| 1500000 | 104.33 ms | 50.88 ms | 42.67 ms |

**Table 3.** ScalaMeter benchmark outputs with a benchmark mapping to Floats

```
[GC (Allocation Failure) 331636K->115092K(1009152K),
   0.0104412 secs]
[GC (Allocation Failure) 385428K->164252K(1009152K),
   0.0112601 secs]
[GC (Allocation Failure) 434588K->172444K(1013248K),
   0.0017833 secs]
[GC (Allocation Failure) 448412K->197004K(1008640K),
   0.0060402 secs]
[GC (Allocation Failure) 472972K->328108K(1007616K),
   0.0329832 secs]
[GC (Allocation Failure) 596396K->344492K(1008128K),
   0.0032716 secs]
[GC (Allocation Failure) 591721K->393652K(998400K),
   0.0084239 secs]
        9. : 685ms (GC : 73ms)
```

(a) ClassTag version

```
[GC (Allocation Failure) 397484K->197233K(1046016K),
   0.0108102 secs]
[GC (Allocation Failure) 525167K->361081K(1046528K),
   0.0241026 secs]
[GC (Allocation Failure) 691468K->557697K(1046528K),
   0.0297168 secs]
[Full GC (Ergonomics) 557697K->328273K(1046528K),
   0.0269909 secs]
        9. : 425ms (GC : 91ms)
```

(b) Old Miniboxed version

```
[GC (Allocation Failure) 408657K->164465K(1045504K),
   0.0138936 secs]
        9. : 325ms (GC : 14ms)
```

(c) New Miniboxed version

**Figure 4.** Results for handmade benchmark mapping over Array-Buffers of 10'000'000 elements to Floats

| Size | ClassTag | Old MbArray | New MbArray |
|---|---|---|---|
| 300000 | 24.62 ms | 11.99 ms | 11.24 ms |
| 600000 | 46.37 ms | 22.94 ms | 22.48 ms |
| 900000 | 57.63 ms | 28.12 ms | 28.43 ms |
| 1200000 | 90.35 ms | 44.57 ms | 43.40 ms |
| 1500000 | 105.27 ms | 49.72 ms | 50.16 ms |

**Table 4.** ScalaMeter benchmark outputs. Mapping ArrayBuffers of Long and Doubles.

### 7.4 JVM Memory

It is also worth noting that the overall speed difference increases at the advantage of the new `MbArray`s when less memory is assigned to the JVM. For example, running any of the ScalaMeter benchmark on a JVM which has less than 400MB completely crashes the version using the old `MbArray`s by throwing an `OutOfMemory` exception. When the amount of memory assigned is between 400MB and 512MB, the old `MbArray` version runs but performs really bad compared to the two other versions, spending a considerable amount of time in the GC. Finally, when more than 512MB is assigned, we obtain the numbers that have been shown so far (Tables 2, 3 and 4) as well as figures 3 and 4), since each of these benchmarks have been ran on a JVM to which was assigned at least 1GB.

### 7.5 Conclusion

In every cases, the version using new `MbArray`s is either faster (tables 2 or 3) than or as fast (table 4) as the version using old `MbArray`s. Futhermore, although it is in some cases running as equivalently fast as the `ClassTag` version – as shows benchmark 2 –, it is actually in most cases faster than the `ClassTag` version, which is demonstrated by benchmarks results of tables 3 and 4.

## References

[1] Scala SIP-15: Value Classes. URL `http://docs.scala-lang.org/sips/completed/value-classes.html`.

[2] A. Genêt, V. Ureche, and M. Odersky. Improving the Performance of Scala Collections with Miniboxing (EPFL-REPORT-200245). Technical report, EPFL, 2014. URL `http://scala-miniboxing.org/`.

[3] J. Gosling. The Evolution of Numerical Computing in Java - preliminary discussion on value classes. URL `http://web.archive.org/web/19990202050412/http://java.sun.com/people/jag/FP.html#classes`.

[4] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7), 2005.

[5] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice (Type constructor polymorfisme voor Scala: theorie en praktijk)*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, 2009. URL `https://lirias.kuleuven.be/handle/1979/2642`. Joosen, Wouter and Piessens, Frank (supervisors).

[6] J. Rose. Value Types and Struct Tearing . URL `https://web.archive.org/web/20140320141639/https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing`.

[7] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA*, 2013.

[8] V. Ureche, E. Burmako, and M. Odersky. Late Data Layout: Unifying Data Representation Transformations. In *OOPSLA '14*. ACM, 2014.