Improving the Interoperation between Generics Translations

Vlad Ureche Milos Stojanovic Romain Beguet Nicolas Stucki Martin Odersky
École polytechnique fédérale de Lausanne, Switzerland
{first.last}@epfl.ch

Abstract

Generics on the Java platform are compiled using the erasure transformation, which only supports by-reference values. This causes slowdowns when generics operate on primitive types, such as integers, as they have to be transformed into reference-based objects.

Project Valhalla is an effort to remedy this problem by specializing classes at load-time so they can efficiently handle primitive values. In its current early prototype[1], the Valhalla compilation scheme limits the interaction between specialized and erased generics, thus preventing certain useful code patterns from being expressed.

Scala has been using compile-time specialization for 6 years and has three generics compilation schemes working side by side. In Scala, programmers are allowed to write code that freely exercises the interaction between the different compilation schemes, at the expense of introducing subtle performance issues. Similar performance issues can affect Valhalla-enabled bytecode, whether the code was written in Java or translated from other JVM languages.

In this context we explain how we help programmers avoid these performance regressions in the miniboxing transformation: (1) by issuing actionable performance advisories that steer programmers away from performance regressions and (2) by providing alternatives to the standard library constructs that use the miniboxing encoding, thus avoiding the conversion overhead.

Keywords generics, specialization, miniboxing, backward compatibility, data representation, performance, Java, bytecode, JVM

1. Introduction

Generics on the Java platform are compiled using the erasure transformation [12], which allows them to be fully backward compatible with pre-generics bytecode. Unfortunately, this also means that they only handle by-reference values (objects) and not primitive types. Thus, primitive values such as bytes and integers have to be converted to heap objects each time they interact with generics. This conversion, known as boxing, compromises the execution performance and increases the heap footprint, forcing Java to lag behind lower-level languages such as C or C++.

[1] As of August 2015 [19, 20].

PPPJ '15, September 08 - 11, 2015, Melbourne, FL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3712-0/15/09…$15.00.
http://dx.doi.org/10.1145/2807426.2807436

The performance drawbacks of erasure are currently being addressed in Project Valhalla[2], an important undertaking led by the Java platform architects and aimed at providing unboxed generics for Java and other JVM languages. The updated bytecode format in Project Valhalla will include the necessary type information to allow load-time class specialization, effectively creating different versions of classes that directly support primitive types. This load-time transformation approach is also employed by the .NET framework [21, 36] in order to implement generics.

Unlike .NET generics, which are always specialized, the current design of Project Valhalla, as of August 2015, makes it an explicit goal to have specialization as an opt-in transformation. This will allow the ecosystem to evolve smoothly from erased to specialized generics, allowing both erased and specialized classes to work side by side. However, in the current early prototype, there are still some restrictions: (1) erased code cannot handle specialized instances in a generic manner and, (2) abstracting over specialized classes using wildcard types [19, 20] pays the cost of boxing primitive types. This is shown in the following example:

```
// The Box class is specialized by virtue of its type
// parameter T being annotated with the "any" prefix:
public class Box<any T> {
  ...
  T getValue() { ... }
}

// The getBoxValue method is compiled with erasure
// since U is not marked with the "any" prefix:
static <U> U getBoxValue(Box<U> box) {
  return box.getValue();
}

// (1) erased code cannot handle specialized class
//     instances (the code will not compile):
getBoxValue(new Box<int>(5));

// (2) abstracting over a specialized class leads to
//     boxing the value (any acts as a wildcard type):
Box<any> box = new Box<int>(5);
System.out.println(box.getValue()); // boxes the value
```

These two code patterns could easily be rewritten to make the code compile and to avoid the overhead of boxing. For the first pattern, adding the `any` prefix to the type parameter `U` of method `getBoxValue` would make it specialized as well, allowing it to handle the incoming argument. In the second pattern, the wildcard, which is equivalent to extending `Box<?>` to value types, could be replaced by the exact type, `Box<int>`, eliminating the overhead of boxing. Yet, in the general case, not all code can be changed at will, due to interactions, backward compatibility or because it resides in external libraries. Thus, a better solution would be to have erased and specialized generics interoperate, ideally without the overhead.

[2] The Valhalla Project is still in its infancy, but early prototypes are openly available and the hard goals have been clearly defined in [18, 25, 26].

The Scala programming language, which also compiles to JVM bytecode, has had compile-time specialization for 6 years [13, 14] and currently has three mechanisms for compiling generics: erasure, specialization and a new arrival, miniboxing [31]. In Scala, all three generics compilation schemes can be freely mixed:

```
// The Mbox class is miniboxed by virtue of the type
// parameter annotation (but could be specialized
// as well, using @specialized):
class Mbox[@miniboxed T](value: T) {
 def getValue(): T = ...
}

// The getMboxValue method is erased:
def getMboxValue[U](mbox: Mbox[U]): U = mbox.getValue()

// (1) erased code can handle specialized instances:
getMboxValue(new Mbox[Int](5))
// (2) programmers can abstract over specializations:
val mbox: Mbox[_] = new Mbox[Int](5)
println(mbox.getValue())
```

Despite the uniform behavior, Scala does pay a hefty price for being able to freely mix code using the three generics compilation schemes: calls between different compilation schemes require boxing primitive values. The reason is that only boxed primitive values are understood by all three transformations. Furthermore, as we will see later on, instantiating a miniboxed (or specialized) class from erased code leads to the erased version being instantiated instead of its miniboxed (or specialized) equivalent, in turn leading to unexpected performance regressions.

In this paper, we show how we completely eliminate the unexpected slowdowns in the miniboxing transformation and, as a side effect, allow programmers to easily and robustly use miniboxing to speed up their programs. The underlying property we are after is that, inside hot loops and performance-sensitive parts of the program, all generic code uses the same compilation scheme, in this case, miniboxing. This way, primitive types are always passed using the same data representation, whether that's the miniboxed encoding (for miniboxing) or the unboxed representation (for specialization).

We show two approaches for harmonizing the compilation scheme across performance-sensitive code:

Issuing actionable performance advisories when compilation schemes do not match, allowing the programmer to harmonize them. For example, when a generic method takes a miniboxed class as a parameter and tries to call methods on it, we automatically generate performance advisories:

```
scala> def getMboxValue[U](mbox: Mbox[U]): U =
     |     mbox.getValue()
<console>:9: warning: The following code could benefit
    from miniboxing if the type parameter U of method
    getMboxValue would be marked as "@miniboxed U":
         mbox.getValue()
             ^
```

Another problem that occurs frequently concerns library evolution: as a new compilation scheme arrives, it is best if all libraries start using it as soon as possible. However, backward compatibility prohibits changing the compilation scheme for the standard library, as it would break old bytecode. In Scala, we had this problem because many of the core language constructs, such as functions and tuples use specialization instead of miniboxing. Similarly, Java has as many as 20 manual specializations for the arity 1 lambda, such as `IntConsumer`, `IntPredicate` and so on. Replacing these by a single specialized functional interface would be desirable, but is realistically impossible. We present a solution for this:

Efficiently bridging the gap between compilation schemes. In the case of miniboxing, which is a compiler plugin, we were not able to change the Scala standard library functions and tuples to use the miniboxing scheme. Instead, we describe the approaches we use to efficiently communicate to the existing library classes, and, where necessary, to replace them by miniboxed equivalents.

With this, the paper makes four key contributions to the Java community and, in the general sense, to the field of compiling object-oriented languages with generics:

- Describing the problems involved in mixing different generics compilation schemes (§2);
- Describing a general mechanism for harmonizing the compilation scheme (§3);
- Describing the approaches we use to fast-path communication between different generic compilation schemes (§4);
- Validating the approach using the miniboxing plugin (§5).

The evaluation section (§5) shows that warnings not only help avoid performance regressions, but can also guide developers into further improving their program's performance.

2. Compilation Schemes for Generics

This section describes the different compilation schemes for generics in Scala. We mainly use Scala for the examples, but the discussion can be applied to Java as well. Differences between Scala specialization and Project Valhalla are pointed out along the way.

2.1 Erasure in Scala

The current compilation scheme for generics in both Java and Scala 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 heap objects. Let us take a simple example, a generic `identity` method written in Scala:

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

When compiled, the bytecode for the method is[3]:

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

As the name suggests, the type parameter `T` was "erased" from the method, leaving it to accept and return `Object`, `T`'s upper bound. 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`.This process, called boxing goes two ways: the argument of method `identity` needs to be boxed while the return value needs to be unboxed back to a primitive type:

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

Boxing primitive types requires heap allocation and garbage collection, both of which degrade 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.

[3] Throughout the paper, we show the source-equivalent of bytecode. The context clarifies whether we are showing source code or compiled bytecode.

2.2 Specialization

Specialization [13, 14] is the second approach used by the Scala compiler to translate generics and, for methods, is similar to Project Valhalla. It is triggered by the `@specialized` annotation:

```
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:

```
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`:

```
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: the reference-based version plus 9 specializations (Scala has the 8 primitive types in Java plus the `Unit` primitive type, which corresponds to Java's `void`). And it gets worse: in general, for N specialized type parameters, it creates 10^N specialized variants, the Cartesian product covering all combinations.

Lacking Project Valhalla's virtual machine support, Scala specialization generates the specialized variants during compilation and stores them as bytecode. This prevents the Scala library from using specialization extensively, since many important classes have one, two or even 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 [31]: 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 C's untagged union. The previous example:

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

Is compiled[4] to the following bytecode:

```
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 boxing in the case of erasure. Yet, our benchmarks on the Java Virtual Machine platform have shown that the miniboxing conversion cost is completely eliminated when just-in-time compiling to native 64-bit code. Further benchmarking has shown that the code matches the performance of specialized code within a 10% slowdown due to coercions [31], compared to a 10x slowdown in the case of boxing.

There is an ellipsis 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 operations such as `toString`, `hashCode` or `equals` to be executed correctly on encoded values:

```
def string[@miniboxed T](t: T): String = t.toString
```

In order to transform this method, we need to treat the primitive value as its original type (corresponding to `T`) rather than a long integer. To do so, we use the type byte:

```
def string(t: Object): String = t.toString
def string_M(T_Type: byte, t: long): String =
                              minibox2string(T_Type, t)
```

Then, when the programmer makes a call to `string`:

```
string[Boolean](true)
```

It automatically gets transformed in the compiler pipeline to:

```
string_M(BOOL, bool2minibox(true))
```

Knowing the type byte, the `minibox2string` can do its magic: decoding the long integer into a "true" or "false" string, depending on the encoded value. Although seemingly simple, the code transformation to implement the miniboxing transformation is actually rather tricky [17, 32, 33].

So far, we have only looked at methods, but transforming classes poses even greater challenges.

2.4 Class Transformation in Project Valhalla

Project Valhalla takes a straight-forward approach to specialization: classes are duplicated and all previous references to the type parameters are transformed. Given the linked list node class:

```
public class Node<any T> {
  T head;
  Node<T> tail;

  public Node(T head, Node<T> tail) {
    this.head = head;
    this.tail = tail;
  }

  public T head() {
    return this.head;
  }

  public Node<T> tail() {
    return this.tail;
  }
}
```

When specializing the `Node` class for the `int` primitive type, Project Valhalla employs its custom classloader to clone and adapt the bytecode for class `Node`. Among other transformations, it replaces references to `T` by `int` [18]:

```
// Node_{T=int} corresponds to Node<int> in the code:
public class Node_{T=int} implements Node_any {
  int head;
  Node_{T=int} tail;
  // ... continued on the next page
```

[4] 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`.

```
  public Node(int head, Node_{T=int} tail) {
    this.head = head;
    this.tail = tail;
  }

  public int head() {
    return this.head;
  }

  public Node_{T=int} tail() {
    return this.tail;
  }
}
```

The load-time translation produces `Node_{T=int}`, which handles unboxed `int` values. The fact that `Node_{T=int}` is not a subclass of `Node` negatively impacts the interoperation with erasure. Let us take an example method, compiled with erasure:

```
static <U> U getNodeTail(Node<U> spec) {
  return spec.tail();
}
```

The corresponding bytecode is:

```
static Object getNodeTail(Node spec) {
  return spec.tail();
}
```

Since `Node_{T=int}` is not a subclass of `Node`, the `getNodeTail` method cannot handle specialized instantiations of `Node` as arguments, forcing the compiler to reject code patterns such as `getNodeTail(new Node<int>(...))`. This limits the interaction between erased and specialized generics and prevents many useful code patterns from being expressed. In turn, it prolongs the time necessary to adopt specialization: if a single generic library is compiled with erasure, its clients need to use erasure as well, otherwise they would not be able to use that library.

Acknowledging the importance of allowing erased methods to handle specialized instances, a solution was later introduced: In the initial prototype, as of December 2014, the only common parent of classes `Node` and `Node_{T=int}` was `Object`. However, as of August 2015, the translation has been improved to automatically introduce the `Node_any` interface which serves as a common supertype of `Node` and its specializations (e.g. `Node_{T=int}`). Aside from simplifying the translation from JVM languages to Valhalla bytecode, this change also enables programmers to abstract over specialized classes:

```
// Node<any> acts as the Node<?> wildcard, except for
// the fact that it also accepts specialized versions:
Node<any> node = new Node<int>(5, null);
System.out.println(node.getValue());
```

This example is translated to the following bytecode:

```
Node_any node = new Node_{T=int}(5, null);
System.out.println(node.getValue()/* of type Object */)
```

The `Node_any` interface is called an "erased view", since it enables accessing the specialized class in a uniform, erased-like manner. Since there is no mechanism to return either a reference or a primitive type (corresponding to the `any` wildcard), the call to `getValue()` boxes the value returned. Thus, the call introduces a silent performance regression, which is the cost of interoperability.

The miniboxing translation allows erased, specialized and miniboxed code to freely interact, enabling both code patterns above, at the expense of adding even more boxing operations.

2.5 Class Transformation in Miniboxing

Scala specialization [13, 14] introduced a better class translation, which is compatible to erased generics. Miniboxing [31] inherited and adapted this scheme, addressing two of its major drawbacks, namely the double fields and broken inheritance. For this reason, we will present the miniboxed class translation scheme directly.

The main challenge of interoperating with erased generics is to preserve the inheritance relation while providing specialized variants of the class, where fields are encoded as miniboxed long integers instead of `Object`s. Let us take the linked list node class again, this time written in Scala:

```
class Node[@miniboxed T](val head:T, val tail:Node[T])
```

The Scala compiler desugars the class to (some aspects omitted):

```
class Node[@miniboxed T](_head: T, _tail: Node[T]) {
  def head: T = this._head       // getter for _head
  def tail: Node[T] = this._tail // getter for _tail
}
```

There are three subtleties in the `Node` translation:

- First, there should be two versions of the class: one where `_head` is miniboxed, called `Node_M` and another one where `_head` is an `Object`, called `Node_L`;
- Then, types like `Node[_]`, which corresponds to Java's wildcard `Node<any>` can be instantiated by both classes, so the two need to share a common interface, the "erased view";
- Finally, this shared interface has to contain the specialized accessors corresponding to both classes (so both classes should implement all the methods).

Given these constraints, miniboxing compiles `Node` to an interface:

```
interface Node {
  def head(): Object    // reference-based accessor
  def head_M(...): long // specialized accessor, which
                        // is not present in the
                        // Valhalla translation
  def tail: Node[T]
}
```

Note that the `tail` method does not have a second version, as it doesn't accept or return primitive values. Then, we have the two specialized variants of class `Node`:

```
class Node_L(_head: Object, _tail: Node) impl Node {
  def head(): Object = this._head
  def head_M(...): long = box2minibox(..., head)
  def tail(): Node[T] = this._tail
}

class Node_M(..., _head: long, _tail: Node) ... {
  def head(): Object = minibox2box(..., head_M(...))
  def head_M(...): long = this._head
  def tail(): Node[T] = this._tail
}
```

As before, the ellipsis corresponds to the type bytes. With this translation, code that instantiates the `Node` class is automatically transformed to use one of the two variants. For example:

```
new Node[Int](4, null)
```

Is automatically transformed to:

```
new Node_M[Int](INT, int2minibox(4), null)
```

And, when `Node` is instantiated with a miniboxed type parameter:

```
def newNode[@miniboxed T](t: T) =
  new Node[T](t, null)
```

The code is translated to:

```
def newNode(t: Object) =
  new Node_L(t, null)
def newNode_M(T_Type: byte, t: long) =
  new Node_M(T_Type, t, null)
```

The translation hints at an optimization that can be done: given a value of type `Node[T]` where `T` is either a primitive or known to be miniboxed, the compiler can call `head_M` instead of `head`, skipping a conversion. The following code:

```
val n = new Node[Int](3, null)
n.head
```

Is translated to:

```
val n = new Node_M(..., 3, null)
n.head_M(...)
```

The rewrite also occurs when the type argument is miniboxed:

```
def getFirst[@miniboxed T](n: Node[T]) = n.head
```

This method is translated to:

```
def getFirst(n: Node): Object =
  n.head           // using reference accessor
def getFirst_M(T_Type: byte, n: Node) =
  n.head_M(T_Type) // using miniboxed accessor
```

At this point, you may be wondering why the `getFirst` bytecode receives a parameter of type `Node` instead of `Node_L`, or, respectively, `Node_M`. The reason is interoperability with erased generics.

2.6 Interoperating with Erased Generics

So far, we have seen the following two invariants:

- we call the `head_M` accessor on receivers of type `Node[T]` where `T` is either miniboxed or is a primitive type, optimistically assuming the receiver is an instance of class `Node_M`;
- otherwise, we call the `head` accessor, assuming the receiver is an instance of class `Node_L`.

Unfortunately, interoperating with erased generics violates both invariants. Consider the following method:

```
def newNodeErased[T](head: T) =
  new Node[T](head, null)
```

During the compilation of this method, using to erased generics, the compiler is forced to make a static (compile-time) choice: Which class to instantiate for the `new Node[T]`?

Since `newNodeErased` can be called with both (boxed) primitives and objects, the only valid choice is `Node_L`, which can handle both cases. Contrarily, `Node_M` can't handle references, since object pointers are not directly accessible in the JVM. Thus, we have:

```
def newNodeErased(head: Object) =
  new Node_L(head, null)
```

Which allows the erased generics to invalidate the invariants:

```
val n: Node[Int] = newNodeErased[Int](3)
n.head // n: Node[Int] => call head_M
```

```
val n: Node = newNodeErased(...)    // retuns a Node_L
n.head_M(INT) // assumption: receiver has type Node_M
```

This way, the call to `head_M` occurs on a `Node_L` instance. The symmetric case can also occur, calling `head` on a `Node_M` class. And, what is worse, we can end up with a `Node_L` class storing a primitive value, which means it will be boxed.

Fortunately, by never promising more than the erased view, `Node`, the compilation scheme is robust enough to handle the mix-up. This allows correctly compiling both patterns in the Valhalla example:

```
def getNodeTail[T](t: Node[T]): Node[T] = t.tail
getNodeTail(new Node[Int](5, null))
val node: Node[_] = new Node[Int](5, null)
node.tail()
```

By producing the following bytecode:

```
def getNodeTail(t: Node): Node = t.tail()
getNodeTail(new Node_M(...))      // Node_M impls Node
val node: Node = new Node_M(...) // Node_M impls Node
node.tail() // call through the Node interface
```

Comparing to Valhalla bytecode, the `Node` interface corresponds to Valhalla's `Node_any` interface, the "erased view". Yet, by replacing references to `Node[T]` by the erased view (instead of the `Node_L` class) miniboxing allows better interoperation.

3. Performance Advisories

The previous section has shown that, when used globally, miniboxed generics provide two key invariants that ensure primitive values are always passed using the miniboxed (long integer) encoding:

- Instantiations of miniboxed classes use the most specific variant (e.g. a value of type `Node[Int]` has runtime class `Node_M`);
- Methods called on a miniboxed class use the most specific specialization available (e.g. a runtime class `Node_M` always receives calls to the miniboxed `head_M` accessor)

The presence of erasure and wildcard-type abstractions (such as `Node[_]`) leads to violations of these two invariants: the reference variant of a miniboxed class may be instantiated in place of a miniboxed variant or the method called may not be the most specific one available. In both cases, the compilation scheme is resilient, producing correct results, at the expense of performance regressions, caused by boxing primitive types.

There key to avoiding these subtle performance regressions is to intercept the class instantiations and method calls that violate the invariant and report actionable advisories to the users, in the form of compiler warnings. Luckily, all the information necessary to detect invariant violations is available during compilation.

3.1 Performance Advisories Overview

Advisories are most commonly triggered by interacting with erased or specialized generics, but can also be caused by technical or design limitations. There are as many as ten different performance advisories implemented in the miniboxing plugin, but in order to focus on the concept, we will only look at the three most common advisories, two of which are caused by the interaction with erased generics. To show exactly how the slowdowns occur, we can take the following piece of code:

```
def foo[@miniboxed T](t: T): T = bar(t)
def bar[@miniboxed U](u: U): U = baz(u)
def baz[@miniboxed V](v: V): V = v
```

The code is compiled to:

```
def foo(t: Object): Object = bar(t)
def bar(u: Object): Object = baz(u)
def baz(v: Object): Object = v
def foo_M(..., t: long): long = bar_M(..., t)
def bar_M(..., u: long): long = baz_M(..., u)
def baz_M(..., v: long): long = v
```

The translation shows that once execution entered the miniboxed path, by calling `foo_M`, it goes through without any boxing, only passing the value in the encoded (miniboxed) representation. Now let's see what happens if the `@miniboxed` annotation is removed:

```
def foo[@miniboxed T](t: T): T = bar(t)
def bar[T](u: U): U = baz(u)
def baz[@miniboxed V](v: V): V = v
```

The bytecode produced is:

```
def foo(t: Object): Object = bar(t)
def bar(u: Object): Object = baz(u)
def baz(v: Object): Object = v
def foo_M(..., t: long): long =
   box2minibox(bar(minibox2box(t))) // boxing :(
def baz_M(..., v: long): long = v
```

Two problems occur here:

- When method `foo_M` is called, it does not have a miniboxed version of `bar` to call further on, so it calls the erased one;
- When method `bar` is called, although `baz` has a miniboxed version, it cannot be called as the type information was erased.

These two problems correspond to the main two performance advisories: forward and backward warnings. A third one, related to data representation ambiguity, will be shown below.

Forward advisories. The first advisory (compiler warning) received by the programmer is also called a forward warning:

```
test.scala:7: warning: The method bar would benefit
   from miniboxing type parameter U, since it is
   instantiated by miniboxed type parameter T of
   method foo:

      def foo[@miniboxed T](t: T): T = bar(t)
                                       ^
```

This advisory pushes the miniboxed representation from caller to callee when the arguments need to be boxed before being passed.

Backward advisories. The miniboxing annotation is also propagated from callee to caller:

```
test.scala:8: warning: The following code could
   benefit from miniboxing specialization if the type
   parameter U of method bar would be marked as
   "@miniboxed U" (it would be used to instantiate
   miniboxed type parameter V of method baz):

       def bar[U](u: U): U = baz(u)
                             ^
```

Ambiguity advisories. Scala allows types to abstract over both primitives and objects. For example, wildcard types (known as existentials in Scala) can abstract over any type in the language. `Any` is the top of the Scala type system hierarchy, with two subclasses: `AnyVal` is the superclass of all value types (and thus primitives) while `AnyRef` is the superclass of all reference types, corresponding to Java's `Object`. Therefore, existentials, `Any` and `AnyVal` are not specific enough to pick a primitive or a reference representation. In this case, we issue a warning and box the values:

```
test.scala:12: warning: Using the type argument "Any"
   for the miniboxed type parameter T of method foo is
   not specific enough, as it could mean either a
   primitive or a reference type. Although method foo
   is miniboxed, it won't benefit from specialization:
          foo[Any](5)
              ^
```

With these actionable warnings, even a novice programmer, not familiar to the miniboxing transformation, can still achieve the same performance as an expert manually sifting through the generated bytecode. We have several examples where programmers achieved speedups over 2x just by following the miniboxing advisories [2, 4, 6]. We will now explain the intuition behind generating performance advisories.

3.2 Unification: Intuition

The reason we chose to present the "forward", "backward" and "ambiguity" advisories is because, although they are only three of the ten cases, they are the warnings a typical programmer is most likely to encounter. They appear in all cases where a specialized variant of either a method or class needs to be chosen:

- Calling a miniboxed method;
- Instantiating a miniboxed class;
- Calling the method of a miniboxed class;
- Extending a miniboxed class or trait;

The one element common to all these cases is the need to pick the best matching miniboxed variant for the given type arguments. For example, given the method `foo` defined previously, for a call to `foo[Int](4)`, the compiler needs to find the best variant of `foo` and redirect the code to it. In this case, since the type argument of method `foo` is `Int`, which is a primitive type, and since the type parameter `T` in the definition of `foo` is marked as miniboxed, it will pick the `foo_M` variant, which uses the miniboxed representation. This operation is called unification, and we have unified the type parameter of `foo`, namely `T`, and a type argument, `Int`. The unification algorithm is also responsible for issuing advisories.

Let us now focus on a more formal definition.

3.3 Unification: Formalization

Let us call the original method or class `O`, with the type parameters `F1` to `Fn` and `VO` the set of specialized variants corresponding to `O`. Each specialized variant `V∈VO` corresponds to a mapping from the type parameters to a representation in the set of {miniboxed, reference, erased}. Let us inverse this mapping, to produce another mapping from type parameters and representations to the specialized variants. Let's call it `VS`.

Then the unification algorithm can be reduced to choosing the corresponding `V∈VO`, for a term of type `O[T1, .., Tn]`. This can be done following the algorithm in Figure 1.

Let us take an example to illustrate this:

```
class C[@miniboxed M, N] // M is mboxed, N is erased
class D[L] extends C[L, Int]
```

When deciding which specialized variant of the miniboxed class `C` to use as class `D`'s parent, we have:

- the original class `O` = `C`;
- the type parameters `F1` = `M` and `F2` = `N`;

- the set of variants `VO` = {`C_M`, `C_L`};
- the inverse mapping `VS` = {`M`: miniboxed and `N`: erased → `C_M`, `M`: reference and `N`: erased → `C_L`}

Now, applying the unification algorithm in Figure 1 for the type parameter `F1` = `M` coupled with the type argument `T1` = `L`, it issues a forward warning followed by outputting (`M`: reference). Then, applying it to `F2` = `N` and `T2` = `Int`, it issues a backward warning and outputs (`N`: erased). From the two bindings, we obtain the specialized variant `C_L` to be a parent of `D`. Indeed, this is what happens in practice:

```
scala> class C[@miniboxed M, N]
defined class C

scala> class D[L] extends C[L, Int]
<console>:8: warning: The following code could benefit
    from miniboxing specialization if the type
    parameter L of class D would be marked as
    "@miniboxed L" (it would be used to instantiate
    miniboxed type parameter M of class C):
       class D[L] extends C[L, Int]
               ^
<console>:8: warning: The class C would benefit from
    miniboxing type parameter N, since it is
    instantiated by a primitive type:
       class D[L] extends C[L, Int]
               ^
defined class D

scala> classOf[D[_]].getSuperclass
res7: Class[_ >: D[_]] = class C_L
```

Now it is easy to guess where the forward and backward names come from: the direction in which the miniboxing transformation propagates between the type parameter and the type argument.

3.4 Unification: Implementation

The performance advisories are tightly coupled with the unification algorithm, which decides the variant that should be used for transforming the code. The processing is done one step at a time, with a type parameter and type argument pair. We will now show some issues that an implementer must be careful about.

Owner chain status. Since methods and classes in Scala can be nested in any order, we must be careful to propagate the status of the type parameters in the owner chain. In the following example:

```
def a[@miniboxed A] = {
  def b[@miniboxed B] = {
    // need to be aware the representation of
    // type parameters A and B when deciding
    // which variant of C to instantiate:
    new C[A, B]()
  }
  ...
}
```

When deciding which miniboxed variant of class `C` to instantiate, we need to be aware of the nested methods we are located in as we duplicate and specialize the code: if we're in method `b_M` inside method `a_M`, we can rely on values of type `A` and `B` to be miniboxed. Contrarily, if we are in method `b` inside method `a`, values of type `A` and `B` are references.

Caching warnings. Instead of issuing warnings right away, they are being cached and later de-duplicated. The reason is that issuing too many warnings diminishes their value. Aside from the three advisories shown, there are special advisories dealing with the specialization transformation in Scala and certain library constructs that we show in the next section. Thus, we define an ordering of advisory priority and, if multiple warnings are cached, we only issue the most important ones.

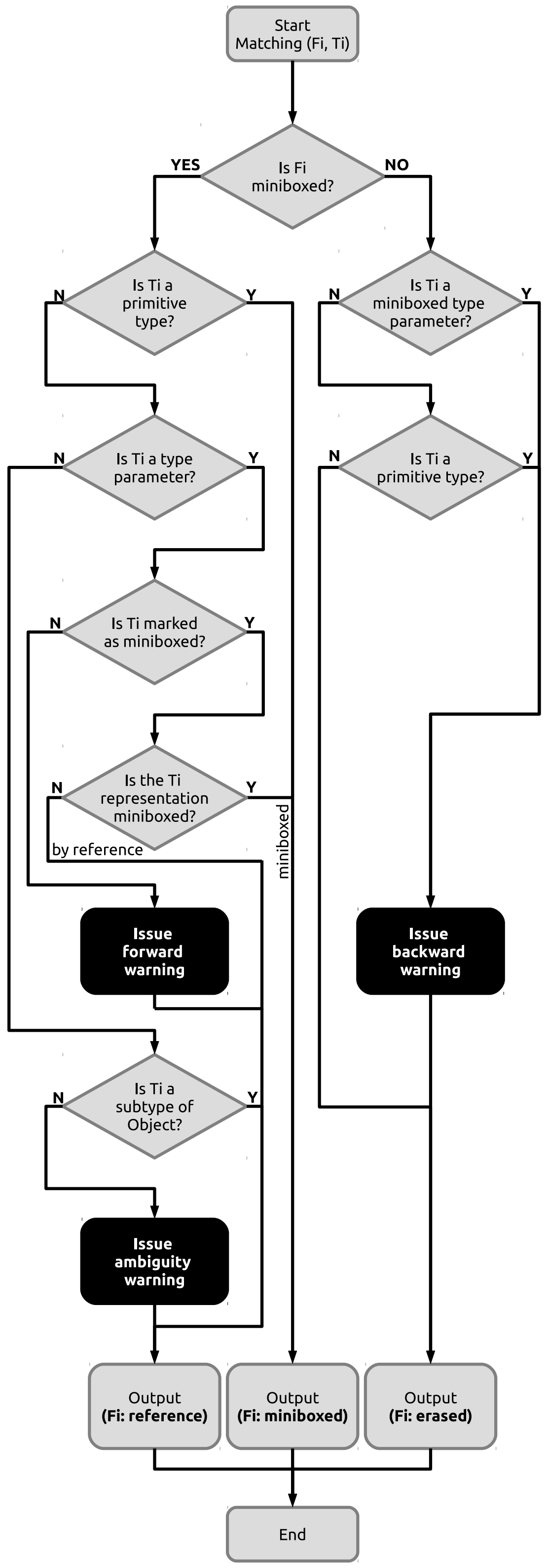

Figure 1: The unification algorithm for picking the data representation of a type parameter.

Suppressing warnings. In certain scenarios, programmers are aware of their sub-optimal erased generic code but, due to compatibility requirements with other JVM programs or due to the fact that code lies outside the hot path, they chose not to change it. In these situations, they need to suppress the warnings, because instead of improving visibility, they might obscure other more important performance regressions in the program. However, a coarse-grained approach such as turning off all warnings is not desirable either, as it completely voids the benefit of advisories. For this scenario, the miniboxing transformation provides the `@generic` annotation, which can suppress performance advisories:

```
scala> def zoo[@miniboxed T](t: T) = t
defined method zoo

scala> zoo[Any @generic](3) // no ambiguity warning
res1: Any = 3

scala> def boo[@generic T](t: T) = t
defined method boo

scala> boo[Int](3)          // no backward warning
res2: Int = 3
```

Libraries. In other cases boxing is caused by the interaction with erased generics from libraries. In this case, the default decision is not to warn, unless the programmer specifically sets the `-P:minibox:warn-all` compiler flag:

```
scala> 3 :: Nil
<console>:8: warning: The method List.:: (located in
   scala-library.jar) would benefit from miniboxing
   type parameter B, since it is instantiated by a
   primitive type:
          3 :: Nil
            ^
res0: List[Int] = List(3)
```

As we will see in the benchmarking section (§5), the performance advisories allow programmers who are not familiar with the transformation to make the same changes an expert would do.

4. Interoperating with Existing Libraries

There is a clear parallel between the manual lambda specializations that are already in the Java Standard Library and thus cannot be eliminated and the specialized constructs in the Scala Standard Library, which cannot be replaced by a compiler plugin. Project Valhalla brings the ability to specialize generics to Java, while miniboxing brings a new compilation scheme for Scala generics. What is common between the two cases is the hard requirement that the new transformations work well with the existing constructs, which use different compilation schemes. This is the problem of interoperating with existing libraries.

In this section we will show how performance regressions occur when miniboxed code interacts with the Scala standard library, which uses either erased generics or the original specialization transformation. To counter these performance regressions, we show three approaches to efficiently bridge the gap between the miniboxing and specialization compilation schemes. Although this section mostly focuses on the interoperation between miniboxing and specialization, the techniques are general and can be applied to Java lambdas and Valhalla as well.

4.1 The Interoperation Problem

When interacting with the library from miniboxed code, the programmers forget the fact that library constructs, such as tuples and functions, do not share the same compilation scheme. Thus, they expect the same performance and flexibility as when using miniboxed classes. However, calling specialized code from miniboxed methods and vice-versa is not easy. For example:

```
def spec[@specialized T](t: T): String = t.toString
def mbox[@miniboxed T](t: T): String = spec(t)
```

The code is translated to:

```
def spec(t: Object): String = t.toString
def spec_I(t: int): String = Integer(t).toString
def spec_J(t: long): String = Long(t).toString
... // other 7 specialized variants
def mbox(t: Object): String = spec(t)
def mbox_M(T_Type: byte, t: long): String = ...
```

The reference-based `mbox` and `spec` methods can directly call each other, since there is a 1 to 1 correspondence. The problem is that, unlike these two methods, none of the specialized variants have a 1 to 1 correspondence to `mbox_M`. This only leaves the reference-based methods as candidates for the direct interoperation between miniboxing and specialization.

Although it may seem like `mbox_M` could directly invoke `spec_J`, since the argument types match, this would be incorrect, as the value `t` in `mbox_M` can be any primitive type, encoded as a long, whereas `t` in `spec_J` can only be a long integer. Thus, if we were to call `spec_J` from `mbox_M` passing an encoded boolean, instead of returning either “true” or “false”, it would return the encoded value of the boolean.

The `mbox_M` method has one more piece of information: `T_Type`, the type byte describing the encoded primitive. In theory, the miniboxed method could use this type byte to dispatch the right specialized counterpart:

```
def mbox_M(T_Type: byte, t: long): String =
  T_Type match {
    case INT  => spec_I(minibox2int(t))
    case LONG => spec_M(minibox2long(t))
    ...
  }
```

Although this indirect approach seems to work and can easily be automated, it is actually a step in the wrong direction: the miniboxing transformation would be introducing extra overhead without offering the programmer any feedback on how and why this happens. Furthermore, when multiple type parameters are specialized, all 10^N possible combinations would have to be added to the match, making it very large. This is likely to confuse the Java Virtual Machine inlining heuristics, causing severe performance regressions.

It may seem like the other way around would be easier: allowing specialized code to call miniboxed methods without performing a switch. However this is not the case because, having been developed first, specialization is not aware of miniboxing. Thus, when calling miniboxed methods, specialization invokes the reference version, boxing the arguments and unboxing the returned value.

With this in mind, our decision was to go with simplicity and symmetry: the bridge between miniboxing and specialization goes through boxing. To allow transparency, miniboxing issues performance advisories about specialized code that should be miniboxed:

```
scala> def mbox[@miniboxed T](t: T): String = spec(t)
<console>:8: warning: Although the type parameter T of
   method spec is specialized, miniboxing and
   specialization communicate among themselves by
   boxing (thus, inefficiently) for all classes ...
```

```
... other than FunctionX and TupleX. If you want to
    maximize performance, consider switching from
    specialization to miniboxing:
        def mbox[@miniboxed T](t: T): String = spec(t)
                                                ^
```

This solution works well with most of the code that lies within the programmer's control, including for the case where 3rd party libraries distribute both a specialized and a miniboxed version. However, the one library which cannot have multiple versions and happens to use specialization is the Scala standard library. The two most wide-spread constructs affected by this are Tuples and Functions, both of which are specialized. This makes the following function a worst-case scenario for vanilla miniboxing:

```
def tupleMap[@miniboxed T,
             @miniboxed U](tup: (T, T), f: T => U) =
  (f(tup._1), f(tup._2))
```

Despite the annotations, with the vanilla miniboxing transformation, all versions of the `tupleMap` method use reference-based tuple accessors and function applications, leading to slow paths irreversibly creeping into miniboxed code. For many applications, this is a no-go, so our task was to eliminate these slowdowns. In the following subsection we present three possible approaches and show where each works best.

4.2 Eliminating the Interoperation Overhead

We show three approaches to eliminating the boxing overhead when calling specialized code from miniboxed classes or methods.

Accessors. The simplest answer to the problem of inter-operating with specialization is to switch on the type byte, as shown previously. To avoid confusing the Java Virtual Machine inlining heuristics, we can extract the operation into a static method, that we call separately. This approach needs to be implemented both for accessors, allowing the specialized values to be extracted directly into the miniboxed encoding and for constructors, allowing miniboxed code to instantiate specialized classes without boxing. This is the approach taken for `Tuple` classes (§4.3);

Transforming objects. The accessors approach allows us to pay a small overhead with each access. This is a good trade-off when the constructs are only accessed a couple of times during their lifetime, which is the case for tuples. In other cases, such as functions, the `apply` method is presumably called many times during the object lifetime, making it worthwhile to completely eliminate the overhead. In this case, a better approach is to replace the `Function` objects by `MiniboxedFunction`s, introducing conversions between them where necessary. This way, the `apply` method exposed by `MiniboxedFunction` can be called directly, and this can compensate for a potentially greater cost of constructing the `MiniboxedFunction` object. This way, switching on the type bytes is done only once, when converting the function, and then amortizes over the function lifetime (§4.4);

New API. In some cases, the API and guarantees are hardcoded into the platform. This is the case for the Scala `Array` class, for which the original miniboxing plugin chose the accessors approach [31]. However, a better tradeoff is achieved by defining a new `MbArray` class with a similar API but different guarantees. This approach will be briefly mentioned in the Arrays subsection (§4.5).

The next sections discuss the three methods above.

4.3 Tuple Accessors

The Scala programming language offers a very concise and natural syntax for library tuples, allowing users to write `(3,5)` instead of the desugared `new Tuple2[Int, Int](3,5)`. Similarly, it allows programmers to write `(Int, Int)` instead of `Tuple2[Int, Int]`. If we were to introduce miniboxed tuples, we would not be able to use the syntactic sugar to express programs, losing the support of many programmers. Instead, a better choice is to efficiently access specialized Scala tuples.

Although we don't have statistically significant data, our experience suggests that `Tuple` classes have their components accessed only a few times during their life. Therefore, both for compatibility reasons and to avoid costly conversions, we decided to allow the `Tuple` class to remain unchanged, instead focusing on providing accessors and constructors that use the miniboxed encoding.

The optimized tuple accessors are written by hand and are explicitly given the type byte:

```
def tuple1_accessor_1[T](T_Tag: Byte, tp: Tuple1[T]) =
  T_Tag match {
    case INT =>
      // the call to _1 will be rewritten to a call
      // to the specialized variant _1_I, which
      // returns the integer in the unboxed format:
      int2minibox(tp.asInstanceOf[Tuple1[Int]]._1)
    ...
  }
```

Once the tuple is cast to a `Tuple1[Int]`, the specialization transformation kicks in and transforms the call to `_1` into a specialized call to `_1_I`, the integer variant. Since the `int2minibox` conversion also takes an unboxed integer, the overhead of boxing is completely eliminated.

The specialized constructors are motivated by two observations: (1) allocating tuples in the miniboxed code without special support requires boxing and, even worse (2) the tuples created use the reference-based variant of the specialized class, thus voiding the benefits of having added tuple accessors. The code for the tuple constructors is also written by hand and is very similar to the accessor code: it dispatches on the type tags to create tuples of primitive types, which specialization can rewrite to the optimized variants.

Introducing accessors and constructors is done by the miniboxing plugin when encountering a tuple access followed by a conversion to the miniboxed representation or when the tuple constructor is invoked with all the arguments being transformed from the miniboxed representation to the boxed one. There are two reasons this step needs to be automated:

- By default, programmers do not have access to the type bytes directly, as this would allow them to introduce unsoundness in the type system (they can inspect their representation using miniboxing reflection, but this is outside the scope);
- One of the reasons tuples are useful is their great integration with the language, allowing a very concise syntax. Asking programmers to use anything other than this syntax would be as bad as developing our own, no-syntax-sugar miniboxed tuple.

With these three changes, benchmarks show a 2x speedup when accessing tuples and a 5% slowdown compared to the equivalent code which accesses the tuples directly. The benchmark we used was a tuple quicksort algorithm (§5).

With the three elements above, accessors, constructors and the automatic code rewriting we create a direct bridge between specialized tuples and miniboxed classes. Unfortunately, as we've seen before, adding such accessors has to be a carefully-weighted, context-specific decision, so automating it would not provide much benefit. For example, this choice would not be suitable for functions.

4.4 Functions

Like tuples, functions in Scala have a concise and natural syntax, which ultimately desugars to one of the `FunctionX` traits, where X is the function arity. For example:

```
val f: Int => Int = (x: Int) => x + 1
```

Desugars to:

```
val f: Function1[Int, Int] = {
  class $anon extends Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  new $anon()
}
```

Since `Function` objects are specialized, the code is compiled to:

```
val f: Function1[Int, Int] = {
  class $anon extends Function1_II {
    def apply_II(x: int): int = x + 1
    def apply(x: Object): Object = // call apply_II
  }
  new $anon()
}
```

When interoperating with miniboxed code, functions can only use the reference-based `apply`, introducing performance regressions.

In our early experiments on transforming the Scala collections hierarchy using the miniboxing transformation [17], we were proposing an alternative miniboxed function trait, called `MbFunction`, and were performing desugaring by hand. The performance obtained was good, but desugaring by hand was too tedious. Later on, we received a suggestion from Alexandru Nedelcu stating that, since functions in Scala are specialized, we should be able to interface directly, thus benefiting from the desugaring build into Scala without paying for the boxing overhead.

Our initial approach used accessors, but we soon learned that switching on as many as 3 type bytes with each function application incurs a significant overhead. Instead, we decided to re-introduce `MbFunctionX` within the code compiled by the miniboxing plugin, where `X` is the arity and can range between 0 and 2 (Scala includes functions with arities up to 22, but arities above 2 are no longer specialized). Yet, this time the `MbFunctionX` objects would be introduced automatically.

Code transformation. The miniboxing plugin automatically transforms `FunctionX` to `MbFunctionX`:

- All references to `FunctionX` are converted to `MbFunctionX`;
- Function definitions create `MbFunctionX` instead of `FunctionX`;

For example, the code:

```
def choice[@miniboxed T](seed: Int): (T, T) => T =
  (t1: T, t2: T) => if (seed % 2 == 0) t1 else t2

val function: Int => Int = choice(Random.nextInt)
List((1,2), (3,4), (5,6)).map(function)
```

Is transformed into:

```
def choice(seed: int): MbFunction2 =
  new AbstractMbFunction2_LL {
    def apply(t1: Object, t2: Object) = ...
    val functionX: Function2 = ...
  }
```

```
def choice_M(T_Type: byte, seed: int): MbFunction2 =
  new AbstractMbFunction2_MM {
    def apply_MM(..., t1: long, t2: long) = ...
    val functionX: Function2 = ...
  }

val function: MbFunction2 = choice_M(...)
List((1,2), (3,4), (5,6)).map(function.functionX)
```

Explaining how the code transformation works is beyond the scope of this paper and has been thoroughly studied in previous literature [32, 33]. The result is that, within miniboxed code, only the `MbFunctionX` representation is used. `FunctionX` is only referenced in a limited number of cases:

- When miniboxed code needs to pass a function to pre-miniboxing code (which uses the `FunctionX` representation);
- When miniboxed code receives a function from pre-miniboxing code (using the `FunctionX` representation);
- When a miniboxed class or method extends a pre-miniboxed entity that takes `FunctionX` arguments;
- When an `MbFunctionX` value is assigned to supertypes of `FunctionX`, it needs to be converted;

Conversions can occur in both directions, from `FunctionX` objects to `MbFunctionX` and back.

Converting `FunctionX` objects to their miniboxed counterparts is done using switches that allow the newly created `MbFunctionX` to directly call the unboxed `apply`, fără boxing:

```
def function0_bridge[R](R_Tag: Byte, f:
    Function0[R]): MiniboxedFunction0[R] =
  (R_Tag match {
    case INT =>
      val f_cast = f.asInstanceOf[Function0[Int]]
      new MbFunction0[Int] {
        def functionX: Function0[Unit] = f_cast
        def apply(): Int = f_cast.apply()
      }
    ...
  }).asInstanceOf[MiniboxedFunction0[R]]
```

In the above code, `f` is statically known to be of type `Function[Int]`, thanks to the type byte. This allows the code to introduce `f_cast`, which in turn allows the specialization transformation to rewrite the call from the reference-based `apply` to the unboxed `apply_I`. On its side, miniboxing instantiates `MbFunction0_M` instead of `MbFunction0` and moves the code to the specialized `apply_M` method. With these rewrites, the anonymous `MbFunction` instance can call the underlying function without boxing:

```
  new MbFunction0_M {
    def T_Type: byte = INT
    // fast path for function application:
    def apply_M(): long = int2minibox(f_cast.apply())
    // fast path for conversion:
    val functionX: Function0 = f_cast
  }
```

Converting `MbFunctionX` objects to `FunctionX` easy, since each `MbFunctionX` object contains its `FunctionX` counterpart in the `functionX` field.

By transforming the function representation, we have eliminated the overhead of calling functions completely. Furthermore, using the previous two strategies to minimize the conversion overhead, we enabled function-heavy applications to achieve speedups between 2 and 14x [1, 33].

Benchmark	Generic	Miniboxed
Builder	161.61 s	53.56 s
Map	98.43 s	49.38 s
Fold	87.98 s	46.14 s
Reverse	27.97 s	33.84 s

Table 1: RRB-Vector operations for 5M elements.

4.5 Arrays

The array transformation [9] is beyond the scope of this paper, but we included it as a good example for using performance advisories.

The `Array` bulk storage in Scala makes certain assumptions that are not compatible with miniboxing, leading to performance regressions in some corner cases. To address this limitation, we introduced a new type of array, dubbed `MbArray`, which integrates very well within the miniboxing transformation. However, since the `MbArray` guarantees do not match the ones offered by Scala arrays, we cannot automate the transformation. Instead, we issue performance advisories to inform programmers about `MbArray`:

```
scala> def newArray[@miniboxed T: ClassTag] =
     |  new Array[T](100)
<console>:8: warning: Use MbArray instead of Array to
    eliminate the need for ClassTags and benefit from
    seamless interoperability with the miniboxing
    specialization. For more details about MbArrays,
    please check the following link:
    http://scala-miniboxing.org/arrays.html
```

This concludes the three approaches to interoperating with the specialized Scala library.

5. Benchmarks

In this section we show three different scenarios where miniboxing has significantly improved performance of user programs. We specifically avoid mentioning benchmarking methodology, as each of the experiments was ran on a different setup. Yet, all three examples show a clear trend: the techniques explained in the paper improve both performance and the programmer experience.

The RRB-Vector data structure [22, 29] is an improvement over the immutable `Vector`, allowing it to perform well for data parallel operations. Currently, the immutable `Vector` collection in the Scala library offers very good asymptotic performance over a wide range of sequential operations, but fails to scale well for data parallel operations. The problem is the overhead of merging the partial results obtained in parallel, due to the rigid Radix-Balanced Tree, the `Vector`'s underlying structure. Contrarily, the `RRB-Vector` structure uses Relaxed Radix-Balanced (RRB) Trees, which allow merges to occur in effectively constant time while preserving the sequential operation performance. This enables the `RRB-Vector` to scale linearly with the number of cores when executing data parallel operations. Thanks to its improved performance, the `RRB-Vector` data structure is slated to replace the `Vector` implementation in the Scala library in a future release.

Benchmark	Generic	Miniboxed some advisories heeded	Miniboxed all advisories heeded
1st run	4192 ms	3082 ms	1346 ms
2nd run	4957 ms	2998 ms	1187 ms
3rd run	4755 ms	3017 ms	1178 ms
4th run	3969 ms	2535 ms	1094 ms
5th run	4073 ms	2615 ms	1163 ms

Table 2: Speedups based on performance advisories, PNWScala

Transformation	Running time
Generic	684.4 ms
Miniboxed (no tuple support)	726.8 ms
Miniboxed (with tuple support)	323.2 ms
Specialized	322.5 ms
Monomorphic	318.1 ms

Table 3: Sorting 1M tuples using quicksort.

The original `RRB-Vector` implementation used erased generics. To show that performance advisories can indeed guide developers into improving performance, we asked a Scala developer who was not familiar with the `RRB-Vector` code base to switch the compilation scheme to miniboxing. Before handing in the code, we removed the parallel execution support [5], reducing the code base by 30%. Then, the developer compiled the code with the miniboxing plugin, which produced 28 distinct warnings. These warnings guided the addition of `@miniboxed` annotations where necessary and the introduction of `MbArray` objects instead of Scala arrays. By following the performance advisories, in less than 30 minutes of work, our developer managed to improve the performance of the `RRB-Vector` operations by up to 3x. A counter-intuitive effect was that it took three rounds of compiling and addressing the warnings before the improvement was visible: each iteration introduced more `@miniboxed` annotations, in turn triggering new warnings, as new methods could benefit from the annotation.

Table 1 shows the performance improvements measured on four key operations of the `RRB-Vector`: creating the structure element by element using a builder and invoking bulk data operations: `map`, `fold` and `reverse`. The ScalaMeter framework [24] was used as a benchmarking harness on a quad-core Intel Core i7-4600U processor running at 2.10GHz with 12GB of RAM, on OpenJDK7.

The numbers show speedups between 1.9 and 3x for the `builder`, `map` and `fold` benchmarks. This can be explained by the fact that, in the erased version, each element required at least a boxing operation, and thus a heap allocation. On the other hand, the `reverse` operation does not require any boxing so there is no speedup achieved. Nevertheless, introducing the miniboxing transformation does not lead to significant slowdowns.

If we consider the `RRB-Vector` development time, which took four months of work and resulted in ~3K lines of source code, the performance advisories issued by the miniboxing plugin allowed a new developer, with no knowledge of the code base, to deploy the miniboxing transformation in a negligible period of 30 minutes, producing speedups of up to 3x.

Image processing. Performance advisories can be used to improve the performance of Scala programs without any previous knowledge of how the transformation works. This was shown at the PNWScala 2014 developer conference [3], where Vlad Ureche presented how the miniboxing plugin guides the programmer into improving the performance of a mock-up image processing library by as much as 4x [4]. The presentation was recorded and the performance numbers are included in Table 2 for quick reference.

Tuple accessors have been tested by implementing a tuple sorting benchmark using a generic quicksort algorithm. Table 3 shows the results for sorting 1 million tuples based on their first element. We used different transformations for the generic quicksort algorithm: first, we benchmarked the erased generics performance, which, as expected, were slow. Surprisingly, the miniboxed version without tuple support was even worse, 7% slower than erased generics. Then, adding tuple accessor support to the miniboxing transformation improved the performance by 2x, making it comparable to the original specialization transformation and to the monomorphic (non-generic, hand specialized) version of the quicksort algorithm.

6. Related Work

The most significant related work lies in the area of run-time profilers which can offer feedback at the language level. We would like to point the work of *St-Amour* on optimization feedback [27] and feature-based profiling [28]. Profiling has existed for a long time at lower levels, such as at the Java Virtual Machine level, with profilers such as YourKit [8] or the Java VisualVM [7] or the x86 assembly, with processor hardware counters.

The area of opportunistic optimizations has seen an enormous growth thanks to dynamic languages such as JavaScript, Python and Ruby, which require shape analysis and optimistic assumptions on the object format to maximize execution speed. We would like to highlight the work of Mozilla on their *Monkey JavaScript VMs [16], Google's V8 JavaScript VM and the PyPy Python virtual machine [10, 11]. While this is just a short list of highlights, the Truffle compiler [34, 35] is now a general approach to writing interpreters that make optimistic assumptions, allowing maximum performance to be achieved by partially evaluating the interpreter for the program at hand, essentially obtaining a compiled program thanks to the first Futamura projection [15].

In the area of data representation, this work assumes familiarity with specialization [13] and miniboxing [2, 31]. The program transformation which enables the functions to be transformed into miniboxed functions is thoroughly discussed in [32, 33]. There has been previous work on miniboxing Scala collections [17] and on unifying specialization and reified types [30]. We have also seen a revived interest in specialization in the Java community, thanks to project Valhalla, which aims at providing specialization and value class support at the virtual machine level [18, 25]. In the Java 8 Micro Edition functions are also represented differently [23].

7. Conclusion

This paper shows several approaches to allowing different generics compilations schemes to interoperate without incurring performance regressions:

- Harmonizing the generics compilation scheme thanks to actionable performance advisories;
- Bridging the gap between library constructs that use different generics compilation schemes, specifically:
  - The accessor approach;
  - The replacement approach;
  - The advisory-based approach.

The implementation results are validated using the miniboxing plugin, which automates the approaches described, showing performance improvements between 2x and 4x.

Acknowledgements

The authors are grateful to the following people who motivated the development of the features described in the paper: Alexandru Nedelcu, Aymeric Genet and Aggelos Biboudis (Functions), Philip Stutz, Stu Hood, Iulian Dragos and Rex Kerr (warnings), Julien Truffaut (tuple accessors). We are thankful to Vincent St-Amour for the ground-breaking work on program optimization advisories.

We would like to thank Brian Goetz, Maurizio Cimadamore and the PPPJ reviewers for their helpful comments and suggestions.

References

[1] ildl Compiler Plugin Documentation. URL `https://github.com/miniboxing/ildl-plugin/wiki`.

[2] The Miniboxing plugin website. URL `http://scala-miniboxing.org`.

[3] PNWScala Conference, . URL `http://pnwscala.org`.

[4] PureImage Library Optimization, . URL `http://scala-miniboxing.org/example_pureimage.html`.

[5] RRB-Vector benchmarks. URL `https://github.com/milosstojanovic/miniboxing-plugin/tree/rrbvector/tests/lib-bench/src/miniboxing/benchmarks/rrbvector`.

[6] Optimistic Respecialization Attempt 6. URL `http://io.pellucid.com/blog/optimistic-respecialization-attempt-6`.

[7] Java VisualVM. URL `https://visualvm.java.net/`.

[8] YourKit Profiler. URL `https://www.yourkit.com/java/profiler/`.

[9] R. M. Beguet. Miniboxing and the MbArray API. Technical report, EPFL, 2015. https://infoscience.epfl.ch/record/208957.

[10] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *ICOOLPS*. ACM, 2009.

[11] C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*. ACM, 2013.

[12] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *OOPSLA*. ACM, 1998.

[13] I. Dragos. *Compiling Scala for Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2010.

[14] I. Dragos and M. Odersky. Compiling Generics Through User-Directed Type Specialization. In *ICOOOLPS*, Genova, Italy, 2009.

[15] Y. Futamura. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 1999. .

[16] A. Gal. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI*. ACM, 2009.

[17] 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/`.

[18] B. Goetz. State of the Specialization, 2014. URL `http://web.archive.org/web/20140718191952/http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html`.

[19] B. Goetz. Announcement, July 2015. URL `https://web.archive.org/web/20150808172443/http://mail.openjdk.java.net/pipermail/valhalla-dev/2015-July/001245.html`.

[20] B. Goetz. Description, August 2015. URL `https://web.archive.org/web/20150808172447/http://mail.openjdk.java.net/pipermail/valhalla-dev/2015-August/001295.html`.

[21] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI*, 2001.

[22] V. U. N. Stucki, P. Bagwell and T. Rompf. RRB Vector: A Practical General Purpose Immutable Sequence. *ICFP*, 2015.

[23] O. Pliss. Closures on Embedded JVM. JVM Languages Summit, Santa Clara, CA, august 2014.

[24] A. Prokopec. ScalaMeter. URL `http://axel22.github.com/scalameter/`.

[25] 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`.

[26] J. Rose. Value Types in the VM, . URL `http://web.archive.org/web/20131229122932/https://blogs.oracle.com/jrose/entry/value_types_in_the_vm`.

[27] V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *OOPSLA'12*, 2012. .

[28] V. St-Amour, L. Andersen, and M. Felleisen. Feature-Specific Profiling. In *CC'15*, 2015. .

[29] N. Stucki. Turning Relaxed Radix Balanced Vector from Theory into Practice for Scala Collections (Master Thesis). Master's thesis, EPFL, 2015.

[30] N. Stucki and V. Ureche. Bridging islands of specialized code using macros and reified types. In *SCALA*. ACM, 2013.

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

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

[33] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating Ad hoc Data Representation Transformations. In *OOPSLA*. ACM, 2015.

[34] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST interpreters. In *DLS*. ACM, 2012.

[35] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Onward!* ACM, 2013.

[36] D. Yu, A. Kennedy, and D. Syme. Formalization of Generics for the .NET Common Language Runtime. In *POPL*, POPL '04. ACM, 2004.