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 promises to remedy this by specializing classes at load-time so they can efficiently handle primitive values.

However, the current design of Project Valhalla severely limits the interaction between erased and specialized generics, disallowing code patterns that seem intuitive for programmers to use. This is a choice likely aimed at preventing programmers from introducing silent performance regressions.

Scala has been using compile-time specialization for 6 years and now has a new generics compilation scheme, miniboxing. In Scala, the interaction between the three compilation schemes is not restricted at all, but can, indeed, introduce subtle performance issues.

In this paper 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 [14], which allows them to be fully backward compatible with pre-generics bytecode. Unfortunately, on the long run, this choice sacrificed performance, since it led to Java primitive types being transformed into 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++.

The performance drawbacks of erasure are currently being addressed in Project Valhalla [20, 26, 27], an important undertaking led by the Java platform architects and aimed at providing unboxed generics for Java and other JVM languages. According to Project Valhalla, the updated bytecode format 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 [22, 37] in order to implement generics.

Unlike .NET generics, which are always specialized, the current design of Project Valhalla, as of June 2015, makes it an explicit goal to have specialization as an opt-in transformation. This would allow the ecosystem to evolve smoothly from erased to specialized generics, allowing both erased and specialized classes to work side by side. However, there are two important limitations in the interaction between erased and specialized generics: (1) erased code cannot handle specialized instances in a generic manner and (2) abstraction over specialized classes is prohibited. This is shown in the following example:

```
1  // The Spec class is specialized by virtue of its type
2  // parameter T being annotated with "any":
3  public class Spec<any T> {
4    String getString() { ... }
5  }
6
7  // The getSpecString method remains erased:
8  static <U> String getSpecString(Spec<U> spec) {
9    return spec.getString();
10 }
11
12 // The following code patterns are prohibited:
13 // (1) erased code handling a specialized class:
14 getSpecString(new Spec<int>());
15 // (2) abstracting over a specialized class:
16 Spec<?> spec = new Spec<int>();
```

There are good reasons to disallow these patterns, as they can silently introduce performance regressions. Still, this makes the language less uniform, surprises programmers and has drawn criticism from the community [2].

The Scala programming language, which also compiles to JVM bytecode, has had compile-time specialization for 6 years [15, 16] and currently has three mechanisms for compiling generics: erasure, specialization and a new arrival, miniboxing [32]. All three mechanisms collaborate and can be freely mixed in the Scala code:

```
1  // The Mbox class is miniboxed by virtue of the type
2  // parameter annotation (but could be specialized
3  // as well, using @specialized):
4  class Mbox[@miniboxed T] {
5    def getString(): String = ...
6  }
7
8  // The getMboxString method is erased:
9  def getMboxString[U](mbox: Mbox[U]) = mbox.getString()
10
11 // The following code patterns are allowed:
12 // (1) erased code handling a miniboxed class:
13 getMboxString(new Mbox<Int>())
14 // (2) abstracting over a miniboxed class:
15 val c: C[_] = new
```

Despite the uniform behavior, Scala does pay a hefty price for being able to freely mix code using the three generics compilation

[Copyright notice will appear here once 'preprint' option is removed.]

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) variants, in turn leading to unexpected performance regressions. We assume this was one of the top reasons that convinced the Project Valhalla architects against allowing erased and specialized generics to work together.

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:

```
1 scala> def getMboxString[U](mbox: Mbox[U]) =
2     |     mbox.getString()
3 <console>:9: warning: The following code could benefit
    from miniboxing if the type parameter U of method
    getMboxString would be marked as "@miniboxed U":
4        mbox.getString()
5            ^
```

Another problem that occurs frequently concerns library evolution: as a new compilation schemes 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:

Offering equivalents of the standard library classes for the new compilation scheme. In the case of miniboxing, which is a compiler plugin, we were not able to change the standard library functions or tuples to the miniboxing compilation scheme. However, we describe a number of approaches that can solve the problem, both manual and automatic.

With this, the paper is making 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 four approaches we used for offering fast-path communication between objects that use 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 the programmer into further improving the program 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, with their implications.

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:

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

When compiled, the bytecode for the method is[1]:

```
1 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`, `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:

```
1 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.

2.2 Specialization

Specialization [15, 16] is a 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:

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

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

```
1 def identity(t: Object): Object = t
2 def identity_I(t: int): int = t
3 def identity_C(t: char): char = t
4 // ... 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`:

```
1 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

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

the method for each type parameter: the reference-based version plus 9 specializations (Scala has the 8 primitive types in Java and `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 [32]: 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 [21]. But unlike a tagged union, miniboxing uses the static type information to avoid carrying tags with each value. The previous example:

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

Is compiled[2] to the following bytecode:

```
1 def identity(t: Object): Object = t
2 def identity_M(..., t: long): long = t
3 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. Yet, our benchmarks on the Java Virtual Machine platform have shown that the miniboxing conversion cost is completely eliminated when compiling the code to native 64-bit assembly code. Further benchmarking has shown that the code matches the performance of specialized code within a 10% slowdown due to coercions [32], 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:

```
1 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:

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

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

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

It automatically gets transformed in the compiler pipeline to:

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

[2] 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`.

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 quite tricky [19, 33, 34].

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:

```
1  public class Node<T> {
2    T head;
3    Node<T> tail;
4
5    public Node(T head, Node<T> tail) {
6      this.head = head;
7      this.tail = tail;
8    }
9
10   public T head() {
11     return this.head;
12   }
13
14   public Node<T> tail() {
15     return this.tail;
16   }
17 }
```

When there is a need for the `Node` class specialized for `int`, the Project Valhalla transformation uses the classloader support to transform references to `T` into `int`:

```
1  public class Node_{T=int} { // Node<int> in the syntax
2    int head;
3    Node_{T=int} tail;
4
5    public Node(int head, Node_{T=int} tail) {
6      this.head = head;
7      this.tail = tail;
8    }
9
10   public int head() {
11     return this.head;
12   }
13
14   public Node_{T=int} tail() {
15     return this.tail;
16   }
17 }
```

The effect of this translation is that the only common superclass of `Node` and `Node_{T=int}` is `Object`. Yet, for an erased generic method, such as:

```
1 static <U> U getNodeTail(Node<U> spec) {
2   return spec.tail();
3 }
```

The generated bytecode is:

```
1 static Object getNodeTail(Node spec) {
2   return spec.tail();
3 }
```

Therefore, the `getNodeTail` method cannot handle any of the specialized variants of class `Node`, unless it is specialized itself. The same occurs when abstracting using wildcard type: `Node<?>` is erased to `Node` and is not compatible with `Node_{T=int}`. While this transformation has the advantage of simplicity, a slightly more complex transformation can achieve compatibility with erased generics.

2.5 Class Transformation in Miniboxing

Scala specialization [15, 16] introduced a better class translation, which is compatible to erased generics. Miniboxing [32] 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 class translation scheme in miniboxing 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<?>` can be instantiated by both classes, so the two need to share a common interface;
- 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 // miniboxed acccesor
  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
```

Gets translated to:

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

The same occurs when a type parameter is miniboxed:

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

This method is translated to:

```
def getFirst(n: Node): Object = n.head
def getFirst_M(T_Type: byte, n: Node) =
   n.head_M(T_Type)
```

At this point, you may be wondering why the `getFirst` method 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:

- `Node_L` stands for linked lists containing references, and we call the `head` accessor on it;
- `Node_M` stands for linked lists containing primitive types and we call the `head_M` accessor on it.

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

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

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

This allows the erased generics to invalidate the invariants:

```
val m = newNodeErased[Int](3)
n.head
```

Which is translated to:

```
val m = newNodeErased(Integer.valueOf(3)) // Node_L
n.head_M(INT)// implicit assumption: class is Node_M
```

This way, the call `head_M` occurs on a `Node_L` class. 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 has to be boxed. While the compilation scheme is robust enough to handle the mix-up, the problem is converting data between representations: from boxed to miniboxed and back. This kills performance.

3. Performance Advisories

The previous section has shown that, when used alone, 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 available (e.g. a value of type `Node[Int]` has runtime class `Node_M`);
- Methods called on a miniboxed class use the most specific variant available (e.g. a runtime class `Node_M` never receives calls to the reference-based `head` 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:

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

The code is transformed to:

```
1 def foo(t: Object): Object = bar(t)
2 def bar(u: Object): Object = baz(u)
3 def baz(v: Object): Object = v
4 def foo_M(..., t: long): long = bar_M(..., t)
5 def bar_M(..., u: long): long = baz_M(..., u)
6 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 from method `bar`:

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

The bytecode produced is:

```
1 def foo(t: Object): Object = bar(t)
2 def bar(u: Object): Object = baz(u)
3 def baz(v: Object): Object = v
4 def foo_M(..., t: long): long =
    box2minibox(bar(minibox2box(t))) // boxing :(
5 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 exactly to the two of the main of performance advisories: forward and backward. 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:

```
1 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:
2
3     def foo[@miniboxed T](t: T): T = bar(t)
4                                         ^
```

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:

```
1 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):
2
3      def bar[U](u: U): U = baz(u)
4                               ^
```

Ambiguity advisories. Scala allows types to abstract over both primitives and objects. A good example are 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` it the superclass of all value (and thus primitive) types 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:

```
1 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:
2            foo[Any]("hi")
3               ^
4 res3: Any = hi
```

With these actionable warnings, even a novice programmer, not familiar to the miniboxing transformation, is still capable of achieving the same performance as a specialization expert manually sifting through the generated bytecode. We have several examples where programmers achieved speedups over 2x just by following the miniboxing advisories [3, 5, 7].

The next section will 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 miniboxed methods;
- Instantiating miniboxed classes;
- Calling method of miniboxed classes;
- Extending miniboxed classes or traits;

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, if we call

`foo[Int](4)`, we need 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, we can call `foo_M`, which uses the miniboxed representation for the argument. The operation we've just done is called unification. And we unified the type parameter of `foo`, namely `T`, and a type argument, `Int`. The unification process chooses the most specific variant and issues 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 examples above 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 `F1` = `N` and `T1` = `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
```

By now you probably guessed 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

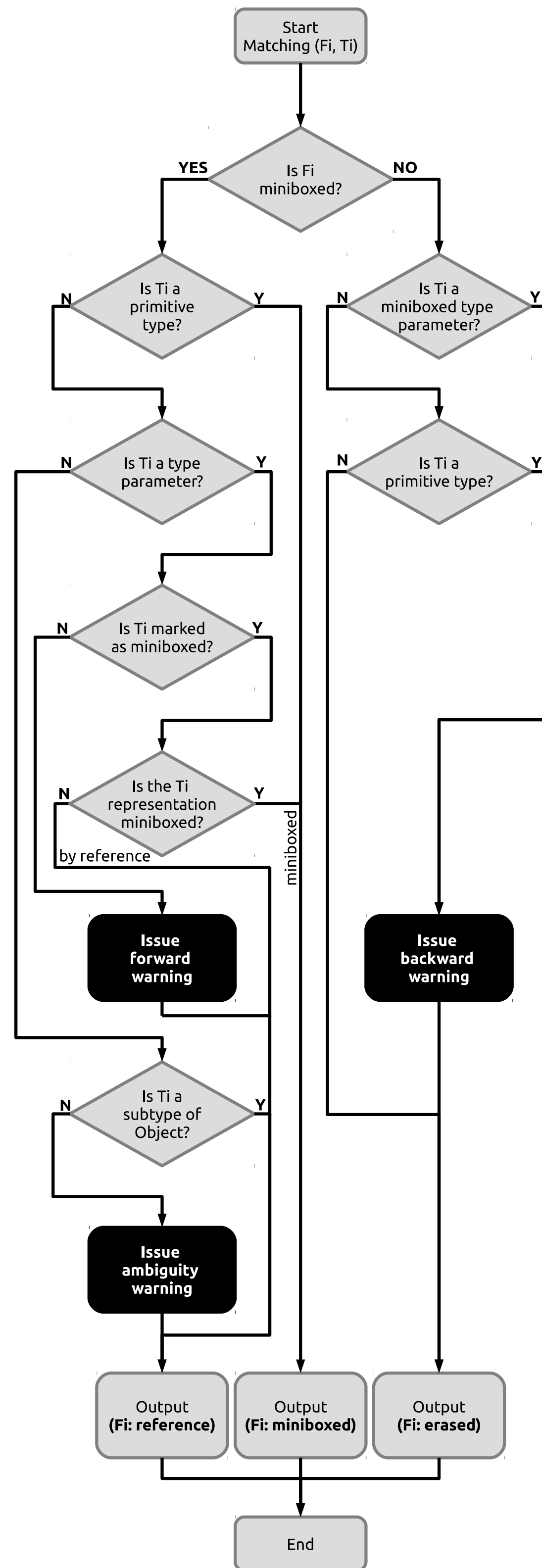

Figure 1: The unification algorithm for picking the data representation of a type parameter.

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 represented as references.

Caching warnings. Instead of issuing warnings right away, they are being cached and later de-duplicated. The reason is that too many warnings quickly lose their value. Aside from the three advisories shown, there are special advisories dealing with the specialization transformation in Scala and certain library constructs we will analyze 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.

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 both forward and backward warnings:

```
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.:: 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 changes an expert would otherwise do.

4. Interoperating with Existing Library Constructs

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 trivial:

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

The translated low-level code is:

```
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. 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 is in possession of one more piece of information: `T_Type`, the type byte describing the encoded primitive type. The type byte allows the miniboxed method to choose which of its specialized counterparts to call:

```
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 back 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 that are difficult to debug.

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 specialization is not aware of miniboxing. Therefore, even when calling miniboxed methods, specialization resorts to invoking the generic 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) on all classes other
   than as 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 `Tuples` (§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 [32]. However, a better tradeoff is achieved by defining a new `MbArray` class with a similar API. 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 pleasant 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 find way to efficiently access specialized Scala tuples, without boxing values.

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 specialized variants.

Closing the cycle is automatically 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 transformation 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:

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

Desugars to:

```
1  val f: Function1[Int, Int] = {
2    class $anon extends Function1[Int, Int] {
3      def apply(x: Int): Int = x + 1
4    }
5    new $anon()
6  }
```

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

```
1  val f: Function1[Int, Int] = {
2    class $anon extends Function1_II {
3      def apply_II(x: int): int = x + 1
4      def apply(x: Object): Object = ...
5    }
6    new $anon()
7  }
```

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 [19], 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:

```
1  def choice[@miniboxed T](seed: Int): (T, T) => T =
2    (t1: T, t2: T) => if (seed % 2 == 0) t1 else t2
3
4  val function: Int => Int = choice(Random.nextInt)
5  List((1,2), (3,4), (5,6)).map(function)
```

Is transformed into:

```
1  def choice(seed: int): MbFunction2 =
2    new AbstractMbFunction2_LL {
3      def apply(t1: Object, t2: Object) = ...
4      val functionX: Function2 = ...
5    }
6  def choice_M(T_Type: byte, seed: int): MbFunction2 =
7    new AbstractMbFunction2_MM {
8      def apply_MJ(..., t1: long, t2: long) = ...
9      val functionX: Function2 = ...
10   }
11
12 val function: MbFunction2 = choice_M(...)
13 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 [33, 34]. 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`, skipping boxing:

```
1  def function0_bridge[R](R_Tag: Byte, f:
     Function0[R]): MiniboxedFunction0[R] =
2    (R_Tag match {
3      case INT =>
4        val f_cast = f.asInstanceOf[Function0[Int]]
5        new MbFunction0[Int] {
6          def functionX: Function0[Unit] = f_cast
7          def apply(): Int = f_cast.apply()
8        }
9      ...
10   }).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 this rewriting, the anonymous `MbFunction` instance can call the underlying function without boxing:

```
1    new MbFunction0_M {
2      def T_Type: byte = INT
3      // fast path for function application:
4      def apply_M(): long = int2minibox(f_cast.apply())
5      // fast path for conversion:
6      def functionX: Function0 = f_cast
7    }
```

Converting `MbFunctionX` objects to `FunctionX` easy, thanks to the `functionX` method.

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.

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, 34].

4.5 Arrays

The array transformation [11] 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 `MbArry` which integrates very well within the miniboxing framework. However, the `MbArray` API does not match the one in Scala arrays, so we cannot automate the transformation. Instead, we use performance advisories to guide the programmers into switching to `MbArray`:

```
1 scala> def newArray[@miniboxed T: ClassTag] = new
     Array[T](100)
2 <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 will specifically avoid mentioning benchmarking methodology, as each of the experiments was ran on a different setup. Yet, all three examples show a clear trend: using the techniques shown improve both performance and the programmer experience.

The RRB-Vector data structure [23] [30] 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, `RRB-Vector` 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 up as we would expect when executing data parallel operations. Thanks to the parallel improvement, 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
Monomorphic	318.1 ms
Specialized	322.5 ms
Miniboxed + Tuples	323.2 ms
Miniboxed	726.8 ms
Generic	684.4 ms

Table 3: Sorting 1M tuples using quicksort.

Implementation of `RRB-Vector` uses erased generics as a translation for generics and in this benchmark we will show how by following the performance advisories given by Miniboxing plugin we can change it to miniboxing translation and improve performance of its operations. Firstly, only methods which are needed for operations used in benchmarks are kept [6] and all unused parts of the implementation were removed. Then, the code is compiled with the Miniboxing plugin which gave to the user 28 distinct warnings. Warnings advised the user where to add `@miniboxed` annotation and where to use `MbArray` instead of `Array` so the implementation can benefit from miniboxing transformation. By following the suggested advice given by compiler, in 3 steps and less than 30 minutes of work, programmer who is not the author of `RRB-Vector` and not familiar with the code, managed to improve the performances of some `RRB-Vector` operations by 50%.

Operations used for benchmarking the miniboxed and generic variants of `RRB-Vector` implementation are creating the `RRB-Vector` using `RRBVectorBuilder` and invoking `map`, `fold` and `reverse` operations on `RRB-Vector`. ScalaMeter framework [25] is used as a benchmark platform and the measurements are conducted using JDK 1.7 on the machine with processor Intel Core i7-4600U CPU @ 2.10GHz x 4 and with RAM of 12GiB. Results of the benchmark are included in Table 1.

The numbers show that for `builder`, `map` and `fold` tests there is a speedup of at least 2x when miniboxing transformation is used which is expected as both boxing and unboxing of the value parameters are present there. On the other hand, `reverse` operation does not require any boxing and therefore there is no any speedup achieved. However, numbers from `reverse` test show that miniboxing does not introduce any noticeable slowdowns as well. To conclude, by following the advises given by miniboxing plugin, the user who does not have any familiarity with the code, for relatively short period of time (if we consider the time needed for development of the `RRB-Vector` which was approximately 4 months and that implementation has ~3K LOC) can improve the performance of some operations by 50%.

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 [4], where Vlad Ureche presented how the miniboxing plugin guides the programmer into improving the performance of a mock-up of a image processing library by as much as 4x [5]. The presentation was recorded and the performance numbers are included in Table 2 for reference.

Tuple accessors have been tested by Milos Stojanovic by implementing a quicksort-based tuple sorting benchmark [8], which sorts tuples based on their first element. The algorithm used is quicksort and the result shows a 2x speed improvement when the accessors are used. The performance is also on par with both specialized and monomorphic code, the slowdown being at most 5%. The results are shown in Table 3.

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 [28] and feature-based profiling [29]. Profiling has existed for a long time at lower levels, such as at the Java Virtual Machine level, with profilers such as YourKit [10] or the Java VisualVM [9] 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 [18], Google's V8 JavaScript VM and the PyPy Python virtual machine [12, 13]. While this is just a short list of highlights, the Truffle compiler [35, 36] 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 [17].

In the area of data representation, this work assumes familiarity with specialization [15] and miniboxing [3, 32]. The program transformation which enables the functions to be transformed into miniboxed functions is thoroughly discussed in [33, 34]. There has been previous work on miniboxing Scala collections [19] and on unifying specialization and reified types [31]. 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 [20, 26]. In the Java 8 Micro Edition functions are also represented differently [24].

7. Conclusion

In this paper we presented several approaches to allowing different generics compilations schemes to interoperate without incurring performance regressions.

8. 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).

References

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

[2] URL http://mail.openjdk.java.net/pipermail/valhalla-dev/2015-January/000477.html.

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

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

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

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

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

[8] Tuple Accessors Pull Request. URL https://github.com/miniboxing/miniboxing-plugin/pull/199.

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

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

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

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

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

[14] 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.

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

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

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

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

[19] 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/.

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

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

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

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

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

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

[26] 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.

[27] 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.

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

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

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

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

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

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

[34] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating Ad hoc Data Representation Transformations. Technical report, EPFL, 2015. http://infoscience.epfl.ch/record/207050.

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

[36] 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.

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