Improving the JVM Bytecode Generated by the Scala Compiler

Parameterised types specialization

Cristian Tălău
EPFL
cristian.talau@epfl.ch

Abstract

In this semester project report I describe my work on designing a new compilation strategy for Scala parameterised types on top of JVM. I will explore existing solutions and propose the *Miniboxing* strategy which has better speed and space performance than the standard erasure implementation, while it generates less bytecode than the full specialization approach. I will also propose a strategy to take advantage of the Java classloading architecture to avoid bytecode explosion by generating specialized code lazily at runtime.

1. Introduction

One of the most important features of the Scala programming language is represented by the parameterised types (or clsses). Among the most common uses are the Scala collections, higher order functions which are desugared into instances of the parameterised type `FunctionN` and tuples.

Although the Java programming language also has this feature, the Java Virtual Machine [5] offers little support for implementing it. In particular, it has different sets of instructions for primitive types and for reference types: e.g. `iastore` for Int, `lastore` for Long and `aastore` for references. The only way to generate code for methods of parameterised typs that handles with all data types is to box primitive values inside objects and generate *erased* code that uses reference-specific instructions. However this boxing and unboxing comes at the cost of space and speed performance. For example, on a 64-bit CPU, the current versionis of JVM would use 256 bits for an instance of `java.lang.Integer` and 64 bits for each reference to it. The caching of integers done inside `Integer.valueOf` can alleviate this problem, but it usually prevents some optimizations like stack allocation of `Integer` objects. Other disadvantages of this approach are the big memory allocation time, poor memory locality, extra level of indirection when accessing the actual value and the unpredictable pauses due to garbage collection.

Another approach to compile parameterised classes, currently implemented in Scala [1], is to generate one version of the code for each kind of datatype. This implementation, while solving the performance problems of the boxing-based one, suffers from the bytecode exposion problem: for a class with N type parameters, it will generate $O(10^N)$ classes.

The *Miniboxing* strategy tries to combine both solutions in order to achieve good space and speed performance with only $O(2^N)$ classes generated. We take advantage of the restrictions that Scala imposes over access to fields (that needs to be made only through setters and getters from outside the class) and use a common representation for primitive types similarly with the way JVM uses the same representation for reference types. While the Java approach boxes/unboxes primitive values to their common reference-based representation, we minibox/miniunbox primitive values to their common representation, using cheaper conversions.

The rest of the report is structured as follows: in the first section I will analyze several existing solutions from the literature with respect to several important design decisions that need to be made when compiling parameterised classes, in the second section I will present some implementation details specific to the Scala programming language and compiler, in the next section I will include some experimental results and finally, I will present some future research ideas and conclusions.

2. Related Work

In this section I will discuss several design decisions that one needs to make when compiling parameterised classes and analyze the solutions present in the literature for other programming languages. Since the problem of parameterised classes is usually handled together with generic methods, I will use *generic code* to refer them.

2.1 What value representations do we use?

2.1.1 Uniform representation

Perhaps the simplest compilation strategy of generic code is to use a common runtime representation for all data types (e.g. as pointers). However, in non-generic code, we may want to store floating point numbers in special registers and object references in other kinds of registers which allow more complex indirect addressing modes. So, we need to have two runtime representations: a uniform one used inside generic code and a specialized one used outside and we need

to convert them back and forth at boundary points. This conversions were introduced by [4] in the context of *ML*-like languages and were called *wrap/unwrap*.

In Java all the values are represented in generic code as object references and outside they use a natural representation [2]. The conversion to the uniform representation works by creating an object to *box* the primitive value. On the other hand, since the natural representation of all objects is a reference anyway, their conversion reduces to just a `checkcast` instruction which is needed by the bytecode verifier ([5], 4.10.2.2).

Another strategy, used in JavaScript engines [6] and possibly used in the future in the JVM [9], represents all the values on 64 (or 128) bits by using techniques like pointer tagging (the lowest bits of a pointer which are 0 due to alignment constraints are reused to store the type tag) and nan-boxing (the 2^{53} bit patterns used to store NaN values are reused for storing other datatypes like integers). Another technique used in JavaScript uses a separate stack to track the type tag of the values. The conversions to the natural representation are usually cheap - just some bit shifting and masking.

2.1.2 Non-uniform representation

At the other end of the spectrum we can use the natural representation also inside the generic code and generate multiple versions of the generic code, one for each possible combination of representations for the values occurring in the code. These versions are called *specializations* of the code (class, method). In the case of generic classes, this means generating B^N classes where N is the number of parameters of the class and B is the total number of representations for some data type. For example, for uniform representation $B = 1$ and for the full specialization used in the Scala compiler [1] $B = |\{$ `Unit, Boolean, Byte, Short, Char, Int, Long, Float, Double, AnyRef` $\}| = 10$. Note that B can also be unbounded as in the heterogeneous translation of the Pizza compiler [8] or in the C++ templates [10] where the class is specialized according to every possible type.

We can already notice that choosing B is one of the main trade-offs between heap memory consumption, speed on the one hand and number of generated classes on the other hand. In our solution we chose to use a uniform representation of all primitive types inspired from the JavaScript one and another representation for reference types. Note that we cannot represent object references on a `Long` because, although it has sufficiently many bits, because object references in JVM are changed by the garbage collector and even if we managed to copy the pointer, it may became unusable after a garbage collector pass.

2.2 When do we generate specialized code?

The approaches in the literature fall into three categories with respect to the time when specialized code is generated: *compile time*, *link time*, *runtime*.

An example of **compile time** approach is the current Scala specialization which, based on user provided annotations generates specialized versions of a parameterised class for type parameters instantiated with primitives. The problem of this approach is that for a class with N type parameters, it generates a $O(10^N)$ versions of it (unless the user specifies a subset of these classes). Usually only a very small number of such specialized version are used in a certain application, so generating them may be avoided by waiting until we really know what the application is going to use.

This brings up the idea to perform specialization at **link time** as done by the C++ templates [10], where generic code is instantiated and type-checked at use site. This approach lacks separate compilation and tends to introduce obscure error messages.

A third option is made possible by the execution environment offered by the virtual machines, namely **run time** specialization, in which parameterised classes are specialized at runtime from a *template* bytecode. This is the approach used in CLR generics [3] where the actual bytecode is used as a template from which the JIT can generate specialized version for instances of generic definitions (classes or methods) and share the code between instances whenever the same representation is used.

A similar approach was tried for JVM in [8] where a classloader was used to generate specialized classes for primitive as well as for reference datatypes. The results reported were negative in part due to the time spent inside the classloader, which at that time did not benefit from the JIT compiler optimizations, and which had to generate classes even for instantiations with non-primitive types. This huge amount of specialized classes can be avoided by specializing only for primitive datatypes. Another problem came from the fact that in Pizza (similarly to Scala) considers primitive types to be instances of `Any` and functions like `hashCode` can be invoked on them. These invocations perform additional boxing operations along with those done at generic code boundary.

Although it seems very promising, there are still some details to solve with run time specialization. First of all we need to choose the right template format from which to generate code at runtime. The main requirement is to allow fast code generation, otherwise any speedup of the specialization would be shadowed by the higher classloading time. A problem specific to the Scala compiler is that generating such a template requires type information that is not present in the backend of the compiler because of the erasure. So, the changes to the compiler are non trivial and in particular, this approach is not suitable for implementing inside a compiler plugin. In [8] the template is a normal class file with annotations specifying a recipe to patch it to obtain the right version for a given combination of type parameters.

Another problem is raised by generic methods which need their code to be ready when the enclosing class is

loaded, so on top of JVM, we cannot wait until the method is actually called to generate the code as the C# VM does. It seems that a quick solution is to move them inside a fresh object which is loaded only when the method is called. However, in the Scala standard library for example, they strongly integrated in the inheritance hierarchy: some generic methods are interface methods which are overridden in subclasses. The solution of the current Scala full specialization is to generate all possible versions for a reasonable number of parameters and fall back to the uniform reference-based representation above that limit [1]. In the Pizza compiler [8], boxing is used for all such methods since they specialize for a possibly unbounded number of types.

A further problem is that we need to be able to run the Scala application even when the specializing classloader cannot be installed in the JVM (for example in application servers like Tomcat). This means that the information that the specializing classloader needs should be encoded as attributes of the class file and which are ignored by a non-specializing classloader.

The *Miniboxing* uses compile time code generation for a smaller number of representation than the current approach. Optionally, by using a custom classloader, it takes advantage of the constant folding optimization implemented in JVM's JIT to be able to avoid the box operations due to invocation of methods like `hashCode` on primitive values.

2.3 Where do we fit specialized classes in the type hierarchy?

In this section, we will assume that we are using the reference based representation for `AnyRef` subtypes, and $B-1 \geq 1$ representations for primitive types, each primitive type having a representation, possibly shared with other primitive types. We also assume that we are generating one version of the parameterised class for each combination of representations for its type parameters. The version generated for the case when all parameters are represented as references is called *the generic version* the others are called *specialized versions*.

First of all, in order to integrate well with the Scala's and JVM's type system, the specialized classes need not be only copies of the generic class where the representation of the type parameters is changed, but they need also to be integrated in the inheritance hierarchy. For example, a code like:

```
case class Foo[T] { def smth : T = ... }
val foo : Foo[_] =
  if (...) Foo[Long] else Foo[String]
```

requires that the specialized class that uses the representation of `T` corresponding to `Long` the generic one (instantiated for `Foo[String]`) have some common supertype: an abstract class, a normal class or an interface. Let's write `FooBase` for the base type of the specialized and generic versions of `Foo[T]`.

It is obvious that it is not realistic to assume that we can generated specialized code for every parameterised class. The motivation are classes like `Function20, Tuple20` which are part of the standard library and which need $O(B^{20}) \geq O(2^20)$ specialized classes to be generated. If we decide not to generate specialized code for a specific class, we generate only the generic version.

Now assume that a method inside a class for which generic code has been generated receives as a parameter an instance of a class for which specialized version have been generated.

```
class Bar[T] {
  def bar(f: Foo[T]) = f.smth
}
(new Bar[Int]).bar(Foo[Int])
```

Since `Bar` is not specialized, the method `bar` handles `f` as if it were the generic class. On the other hand, this method must be able to receive any of the specialized versions as a parameter. So, it means that the methods of the generic class can be invoked on the specialized classes also and provide the same effect as if they were invoked on the generic class. This means that there must exist a supertype of all specialized versions of the class `Foo` containing the methods in the generic class. Let's call it `FooGenericBase`. Another implication is that inside generic code we should replace consider the expressions of type `Foo[T]` as having type `FooGenericBase`. This is the same concept as the generic interface from [8].

Since some of the classes are not specialized, it turns out to be difficult to generate code that always instantiates the specialized version with the correct representation, especially the instantiation is done from generic code. In [8] the authors make use of reflection to accomplish this at the cost of execution time. Another choice is to instantiate the generic class whenever we do not know statically the right representation. For this reason, an expression with static type `Foo[Int]` can evaluate either to an instance of the specialized version corresponding to `Int's` representation or to an instance of the generic class. So, symmetrically to the above situation, there must be a supertype for each specialized class that it shares with the generic version. This supertype allows methods from the specialized versions to be invoked on the generic class. As we have exponentially many specialized versions of the parameterised class, we have, at least conceptually, exponentially many such supertypes (but they may coincide). Let's call them `Foo*Base` where `*` stands for some name mangling schema. Again, as above, we need to replace uses of `Foo[Int]` with the corresponding `Foo*Base` super type everywhere in the specialized code.

Note that a similar problems occur in presence of contravariant types, where `Foo[Long]` is supertype of `Foo[Any]`, so a generic class may be encountered as the result of the evaluation of an expression with static type `Foo[Long]`.

2.3.1 Choosing common supertypes

If we choose to make the super types distinct, then the generic class should implement as many interfaces as there are specialized classes, which means B^N and that these super types must be interfaces due to single inheritance restriction of the JVM. Note that even if we use runtime generation for the specialized classes, the generic class must implement the interface that corresponds to each of them since interfaces cannot be added to an already loaded generic class, as more specialized versions are loaded.

In the rest of the section we will focus on the case where all the supertypes coincide and analyze the three options for materializing this supertype: *interface*, *abstract class*, *concrete class*. We will use as a running example the following class hierarchy, that contains most of the interesting cases:

```
class Bar[T] extends Base
class Foo[T] extends Bar[T]
class Baz extends Bar[Int]
```

Let's denote by `Foo$S, Bar$S` the supertypes and by `Foo$P, Bar$P` the specialized versions that use the representation that corresponds to `Int`. The generic classes will be just `Foo, Bar`.

Conceptually, each specialized version of a parameterised class will contain m methods that are adapted to work with its specific representation for its type parameters. These methods are called the *original methods* of the specialized class and the class it is called their *original owner*. The original methods of the generic version of the class are called *the generic methods*. The supertype will contain $B^N \cdot m$ methods, that is the collection of all the original methods of the specialized classes.

Note that, as an optimization, two different specialized may share methods if the methods operate only on values that have as type the type parameters that use the same representation in both classes. However, in this presentation, we do not take this optimization into account.

Interface The resulting class hierarchy looks like in Figure 1. Since the interface has $B^N \cdot m$ methods, so do all the specialized classes. Each of the specialized classes will provide an implementation for its m original methods. The rest of the methods will just convert the argument and forward to the original methods. From every specialized class only 2^N methods can be ever called (corresponding to reference-based representation or the specialized representation of its type parameters). The rest of the methods can just throw an exception or can be left unimplemented and the JVM will throw an exception for us if, because of some bug in the compiler, they happen to be called.

A limitation for this approach is that JVM allows only public methods inside interfaces while our classes may have private methods that need to be specialized as well.

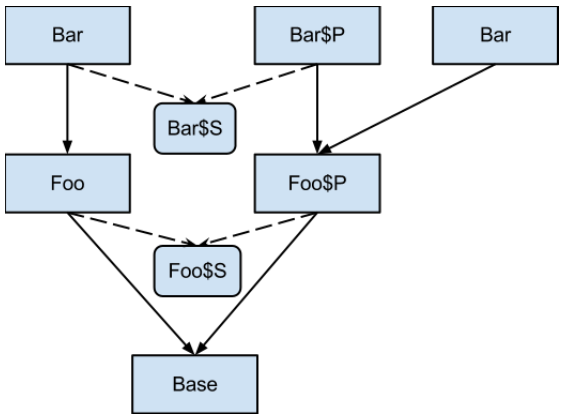

Figure 1. Class hierarchy for interface supertype

Abstract class The resulting class hierarchy looks like in Figure 2. The improvement over the *interface* approach is

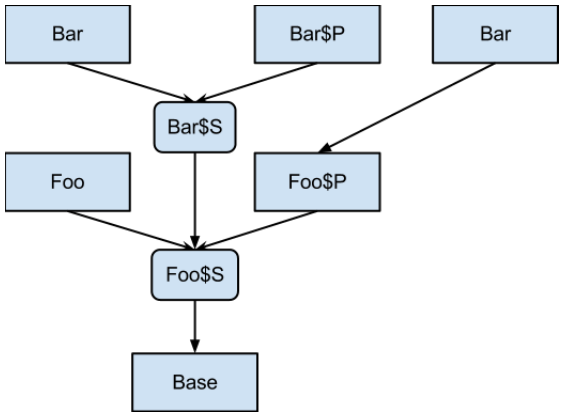

Figure 2. Class hierarchy for abstract class supertype

that the abstract class can provide default implementation for all methods to just forward to the original ones. This way, the specialized classes need to implement only $2m$ methods: the original ones by adapting the body from the original class, and the generic ones to forward to them. All the other methods will forward to the generic ones which in turn forward to the original ones. A small penalty is thus encountered if neither the original nor the generic method is called.

Because of the JVM single inheritance restriction the specialized versions of `Bar` do not extend the corresponding specialized versions of `Foo`. In order to simulate the inheritance, the specialized methods from `Foo` need to be copied, augmenting the size of the class `Bar`. However, the increase in the bytecode size is only linear with the length of the inheritance chain.

This approach solve the problem with specialized private methods because abstract classes are allowed to have private members. Also, in the case of an abstract

class supertype, the specialized methods will be invoked with `invokevirtual` which is usually faster than `invokeinterface`.

Concrete class In this case, the super type must be one of the specialized versions of the parameterised class. We chose it to be the generic class and the resulting class hierarchy looks like in Figure 3. As there is no

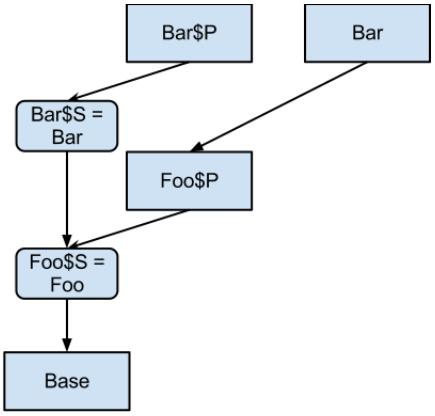

Figure 3. Class hierarchy for concrete class supertype

notable difference in bytecode size compared to the abstract class approach, the most important advantage of using the generic class as a supertype is its simplicity: we don't have to replace every occurrence of `Foo[X]` with the supertype as described above since they are actually the same.

One disadvantage of this approach is the extra heap space used because a specialized class inherits the fields generic class and it needs another `Boolean` field to avoid executing both the specialized and generic version of the constructor. One further problem is that some private fields of the generic class need to be accessible from the specialized subclasses which violates the JVM security model.

Note that in all cases, whenever we do not use the natural representation for a type parameter of a parameterised class, we need to add add bridge methods in subclasses non-parameterised subclasses.

Despite the fact that the last two approaches generate less bytecode, at runtime, the size of the *vtables* will be the same for all three approaches, namely $B^N \cdot m$.

2.4 Implementation specialization

In the previous sections, we discussed different strategies to specialize the *interface* (i.e. the method signatures) of a parameterised class and the restrictions that it must satisfy in order to integrate nicely with the type hierarchy. The purpose was to lower the conversion cost at class boundary, while keeping under control the number of the methods in a specialized class.

While in some programming languages like Haskell, the values whose type is a type parameter can only be used as black boxes and only passed around and stored, in Scala we can perform a richer set of operations on them: invoke methods from `Any`, store them into an array (which on JVM requires different opcode depending on the type of the value), etc. These operations usually require conversions to the natural representation of the values by pattern matching on their type at runtime.

It means that it makes sense to have specialized versions of a parameterised corresponding to $B' > B$ representations, even if the interface was specialized for only B representations. The advantage of this further specialization is that at runtime, only the used specialized versions will use memory and they can even be generated lazily. On the other hand, if we have $B^N \cdot m$ methods in the interface, the *vtable* will be that big irrespective to the number of specialized versions that are actually used.

I will present the implementation specialization idea on the following example:

```
def upd0[T](arr: Array[T], e: T) =
  arr(0) = e
upd0(Array(0, 1, 2), 3)
```

where the method `upd0` needs to perform a pattern match at runtime on the runtime type of `arr` since for different types of arrays, it needs to use a different opcode to store the value `e` (which has to be unboxed by another pattern match on its type). If we had an implementation specialized for `Int`, the above code can be rewritten to:

```
def upd0$Int[T](arr: Array[T], e: T) =
  arr.as[Array[Int]](0) = e.as[Int]
upd0$Int(Array(0, 1, 2), 3)
```

and the pattern match becomes just a `checkcast` instruction.

Moreover, if the method `upd0$Int` is inlined, then the boxing operation of `e` before the method call and its unboxing inside the method will appear close to one another in the resulting bytecode and they can be both removed by a simple peephole optimization, or after some escape analysis, the compiler may decide to stack allocate `e` and to reduce the garbage collection cost.

3. Miniboxing

We decided to use a uniform representation for all 9 Scala primitive types: `Unit`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double`. Since all of them use at most than 64 bits, we chose to encode them on a `Long`. However for invoking methods like `toString`, `##`, storing values in an array, converting to reference based representation, etc. we need their type at runtime. Since we have only 9 primitive types, it is enough to represent them as a `Byte` tag.

The good news is that the tag is determined by the type parameter of the class, and is constant for every class in-

stance. So, it does not need to be passed around together with the `Long` represented value, but only at instance creation time, and to be stored in a special synthetic field.

We specialize the class interface and implementation at compile time for all possible value representations. Since each type parameter can be either reference type and represented after erasure as `Object`, or primitive and represented as `Long` we have 2^N classes for N type parameters. At class loading time, we optionally further specialize the implementation of the classes that use primitive values to speed up operations that depend on the type tag.

We chose to use a single interface as the common supertype for all specialized classes because of the nicer semantic properties when we extend a specialized superclass (in the other two approaches, we extend tin fact the generic version). However, the other two approaches are applicable as well.

The Scala operations that depend on the type tag of a *Miniboxed* value include array access on an array whose elements have the type given by a tag, and invocation of methods inherited from `Any` on a primitive value with the type given by a tag. Note that we do not specialize generic methods, so the generic version is called always irrespective to the type tags.

Only one more operation involves using the type tags, namely instance creation. The problem is trivial whenever the parameters are known types or type parameters of the enclosing class. In this case, we just invoke the corresponding constructor. However, if the type parameters are not specialized, but only have a tag associated, we know which constructor to use only at runtime, when the value of the tag is known. This can be implemented either with reflection or with a pattern match on the tags of the type parameters as long as they are reasonably few (one or two). Another approach is to use a classloader to generate on the fly a factory that will at its turn generate the instance.

Since all these solutions incur a significant overhead for instance creation, we chose to instantiate the generic class in such cases.

With this simplification, we can see that the operations that depend on the type tag can be isolated as methods inside the `MiniboxTagDispatch` object. Since the body of the methods in the specialized classes is only changed to use these tag-dispatch methods instead of the original ones, the increase in the size of code is only linear with a small factor. For performance reasons, we decided to mark these methods as `@inline`.

3.1 Prototype compiler plugin

We implemented [7] a prototype for the *Miniboxing* idea as a compiler plugin. Being still a prototype it contains only a part of what is described above. Based on a `@minispec` annotation, it specializes parameterised types using the `Long` based representation when specializing for primitive values. For the inheritance hierarchy, we use the interface-based approach except that only the generic class is used as superclass to avoid generating additional bridge methods. We also assume that our classes have only public methods.

Another tricky case is represented by the arrays which always need to use the natural representation of their elements because converting them at class boundary would be very expensive. For this reason, their support in the current implementation is subject to several restrictions. On the other hand, fields can escape from a class only via getters and setters, so we can use our `Long`-based representation for them.

As in the full specialization, the compilation is split into three steps. The first one takes care of creating specialized version of the parameterised class and to fit them in the type hierarchy. The second step is to replace the instantiations of the generic class with instantiations of specialized versions whenever we have sufficient information available about its type parameters. The third step consists of rewiring method calls to use the methods that would require the least number of conversions (again, when we have sufficient information about that). For the tests that we have made, we used our prototype to perform the first step.

The most complicated part in implementing this in the Scala compiler is that these specializations are easiest to understand and express in terms of representations of values. i.e. `Object` or `Long`. However, in the compiler the conversion (at least for `Object`) takes place during the erasure phase while specialization happens several phases before. Between these phases we insert casts in the body of specialized methods that are guaranteed to be identity only after erasure.

3.2 Dynamic specialization

As noted above, since the operations like array access depend on the type tag, it would make sense to specialize the implementation of a parameterised type more than its interface.

What this specialization does is to replace the tag-dispatch methods by the operation corresponding to the runtime value of the type tag. This can be thought of as an example of *constant folding* on type tags that replaces the pattern matching with the correct branch. The problem is that type tags are constant for each instance of a class, and the body of the method is shared by all instances, so we cannot fold the constants into it. However if we duplicate the class for each possible value of the tag and make the tag field `static final` in that class, the JIT will fold it inside the body of the class. Note that this duplication is very similar with *procedure cloning* and that our optimization is an instance of *interprocedural constant folding*.

This method of specialization can be used inside the classloader that can duplicate the classes easily and with the cooperation of the JIT compiler which performs the local constant folding.

A prototype classloader has been implemented that makes the tag fields static final so that they can be constant folded by the JIT.

4. Evaluation

In order to asses the performance of the Miniboxing approach we developed several micro-benchmarks involving a linked list and a growable vector.

For the linked list we test *insertion* of N elements, computing a *hashCode* by combining the hash codes of elements, and *lookup* of 1% of the elements in a sequential fashion. For the array we test also the *insertion* of N elements, the *reverse* operation, and the *lookup* of 1% of the elements. The datastructures were instantiated with `Int` and `Double` as the first one has a very cheap conversion while the latter has the most expensive conversion to the `Long` representation among all primitive types.

We use use $N = 1,000,000$ elements, and we iterate each operation several times in order to obtain a time of approximately 1 second. We measured the time only after the VM was warmed-up and JIT compiled everything by skipping the first few runs of each operation. We also forced a full garbage collection cycle before starting the time measurement.

Five compilation strategies have been compared: *erasure* - the default strategy used in Scala, *full specialization* - available in Scala with a special annotation, *Miniboxing*, *Miniboxing optimized* - the tag-dispatching methods are inlined, *Miniboxing with ClassLoader* - which uses a specializing classloader in addition to the prvious strategy.

We conducted the measurements on a 64-bit Intel Core 2 Duo @ 2.5GHz computer, and JVM 1.6.0_26. The results are summarized in Table 2.

We can notice that for some cases, the miniboxing approach was better than the full specialization which uses natural representation for primitive values. This can happen because the full specialization do not specialize methods that do not have type parameters in their signature. In consequence, the nullary *hashCode* will box/unbox the value stored in the head of the list.

Another observation is that for operations that need runtime type information (like accessing an array) the classloader approach is up to 4x faster.

In Table 1 the heap space occupied by *payload* of the linked list is shown. By p we denote the pointer size - 4 or 8 bytes, by o the overhead of an object (two machine words for *klass* and *mark* plus padding [9]) and by s the size of the primitive data type. For the full specialization, the p term comes from the fact that we inherit the fields of the generic class. In the miniboxing with classloader approach, we get make the type tag `static`, so, we do not have to keep it around in every instance. It can be seen that on 64-bit processors, even for boolean payload, the miniboxing approach uses less memory than the other approaches.

erasure	$N \cdot (p + o + s)$
full spec	$N \cdot (p + s + 1)$
MB	$9N$
MB+CL	$8N$

Table 1. Heap space comparison

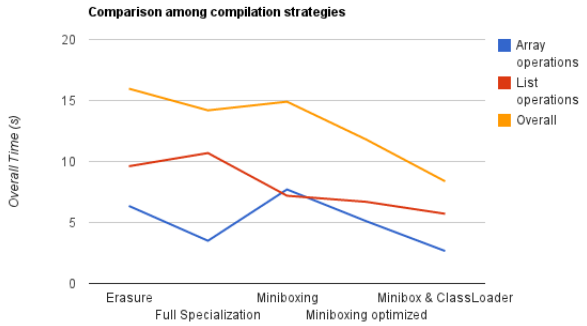

Figure 4. Overall speed performance

5. Future work

An interesting idea that we did not explore sufficiently is that instead of creating multiple methods that receive their parameters according to each combination of representations, we can use an single method that has one parameter for each parameter of the original method and each representation. For example:

```
def foo(x: T, y: u)
```

translates to:

```
def foo(x$P: Long, x$R: Any,
        y$P: Long, y$R: Any)
```

The implementation of such a method, when specialized for primitive types will use only the parameters with primitive representation. When calling such a method we pass some dummy value in the parameters that are not going to be used by the implementation. This may reduce the exponential number of methods inside the supertype.

Another point that is far from being solved by *Miniboxing* is the compilation of generic methods.

Finally, a full implementation of the *Miniboxing* idea would allow us to run some macro-benchmarks and to compare it more accurately to the existing implementations.

Acknowledgments

I would like to thank to Vlad Ureche and Miguel Garcia for the close collaboration and insightful discussions. I would also like to thank Martin Odersky and Iulian Dragos for the feedback given during the presentation.

	Parameter	erasure	full spec	MB	MB opt	MB+CL
array insert	`Int`	1218	855	1039	952	515
	`Double`	1544	907	1592	1300	960
array reverse	`Int`	994	918	1226	726	406
	`Double`	468	430	1278	1058	391
array lookup	`Int`	1490	100	1197	446	116
	`Double`	644	291	1378	630	273
array bytecode	`Int`	3.8K	27.9K	16.1K	19.7K	19.7K
	`Double`					
list insert	`Int`	1543	1913	1509	1478	898
	`Double`	2181	2442	1111	982	849
list hashCode	`Int`	1244	1480	1279	1142	959
	`Double`	1510	1750	1243	1037	1057
list lookup	`Int`	1464	1478	1045	1077	943
	`Double`	1669	1629	1005	979	1008
list bytecode	`Int`	2.1K	13.4K	7.6K	10.5K	10.5K
	`Double`					

Table 2. Time and bytecode size comparison between the five approaches

References

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

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java (TM) Language Specification*. Addison-Wesley Professional, 2011.

[3] A. Kennedy and D. Syme. Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices*, volume 36, pages 1–12. ACM, 2001.

[4] X. Leroy. Efficient data representation in polymorphic languages. In *Programming Language Implementation and Logic Programming*, pages 255–276. Springer, 1990.

[5] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The java virtual machine specification. *Addison Wesley*, 2012.

[6] D. Mandelin. Jagermonkey: the 'halfway' point. `http://blog.mozilla.org/dmandelin/2010/05/10/jm-halfway/`.

[7] Miniboxing-Plugin. Source repository. `https://github.com/miniboxing/miniboxing-plugin/tree/ctalau`.

[8] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your pizza—translating parameterised types into java. *Generic Programming*, pages 114–132, 2000.

[9] J. Rose. fixnums in the vm. `https://blogs.oracle.com/jrose/entry/fixnums_in_the_vm`.

[10] D. Vandevoorde and N. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201734842.