Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations

Vlad Ureche Cristian Talau Martin Odersky
EPFL
{first.last}@epfl.ch

Abstract

Parametric polymorphism enables code reuse and type safety. Underneath the uniform interface exposed to programmers, however, its low level implementation has to cope with inherently non-uniform data: value types of different sizes and semantics (bytes, integers, floating point numbers) and reference types (pointers to heap objects). On the Java Virtual Machine, parametric polymorphism is currently translated to bytecode using two competing approaches: homogeneous translation requires boxing, and thus introduces indirect access delays. Heterogeneous translation duplicates code and adapts it to each value type individually, inflating the bytecode size. Therefore speed and size are at odds with each other. This paper proposes a novel translation which improves the situation by significantly reducing the bytecode size without affecting the execution speed. The key insight is that all non-pointer types can be encoded on long integers, reducing the need to duplicate code for each value type. The resulting bytecode approaches the performance of monomorphic code and obtains speedups of up to 22x over the homogeneous translation, all with modest increases in size.

Categories and Subject Descriptors D.2.3 [*Language Constructs and Features*]: Polymorphism; E.2 [*Object representation*]

General Terms data representation, specialization, scala, java virtual machine, bytecode

Keywords miniboxing

1. Introduction

Parametric polymorphism allows programmers to describe algorithms and data structures irrespective of the data they operate on. This enables code reuse and type safety. For the programmer, generic code, which uses parametric polymorphism, exposes a uniform and type safe interface that can be reused in different contexts, while offering the same behavior and guarantees. This increases productivity and improves code quality. Nowadays most programming languages offer generic collections, such as linked lists, array buffers or maps as part of their standard libraries.

But despite the uniformity exposed to programmers, the lower level translation of generic code struggles with fundamentally non-uniform data. To illustrate the problem, we can analyze the `contains` method of a linked list parameterized on the element type, `T`, written in the Scala programming language:

```
def contains(element: T): Boolean = ...
```

When translating the `contains` method to lower level code, such as assembly code or bytecode targeting a virtual machine, a compiler needs to know the exact types of the arguments, so they can be correctly retrieved from the stack, registers or read from memory. But since the list is generic, the type parameter `T` can have different bindings, depending on the context, from a byte or an integer, to a floating point number or a pointer to a heap object, each with varying sizes and semantics. So the compiler needs to bridge the gap between the uniform interface and the non-uniform low level implementation.

Two main approaches to compiling generic code are in use today: heterogeneous and homogeneous. Heterogeneous translation duplicates and adapts the body of a method for each possible type of the incoming argument, thus producing large binaries. On the other hand, homogeneous translation generates a single method but requires data to have a common representation, irrespective of its type. This common representation is usually chosen to be a heap object passed by reference, leading to indirect access to values. This, in turn, slows down the program execution and increases heap usage. The conversions between value types and heap objects are known as boxing and unboxing. A different uniform representation, typically reserved to virtual machines for dynamically typed languages, uses the fixnum [21] representation. This representation can encode different types in the same unit of memory by reserving several bits to encode the type and using the rest to store the value. Aside from reducing value ranges, this representation also introduces delays when dispatching the corresponding operation depending on the type. An alternative is the tagged union [7], which does not restrict the value range but requires more storage space.

Recognizing the need for a tradeoff between execution speed and binary size, Scala specialization [4] allows an annotation-driven, compatible and opportunistic transformation to Java Virtual Machine bytecode [10]. Programmers can explicitly annotate generic code to be transformed using a heterogeneous translation, while the rest of the code is translated using boxing [3]. Specialization is a compatible transformation because specialized and homogeneously translated bytecode can be freely mixed. For example, if both a generic call site and its generic callee are specialized, the call will use primitive values instead of boxing. But if either one is not specialized, the call will fall back to using boxed values. Specialization is also opportunistic in the way it injects specialized code into homogeneous one. Finally, being annotation-driven, it lets programmers decide on the tradeoff between speed and code size.

Unfortunately the interplay between separate compilation and compatibility forces specialization to generate all heterogeneous variants up front instead of delaying their instantiation to the time they are used, like in C++ [20]. Although in some libraries this might be desirable [2], generating all heterogeneous variants up front means specializing must be done cautiously so the bytecode generated does not explode. To give a sense of the amount of bytecode specialization produces, for the Scala programming language which has 9 value types and 1 reference type, fully specializing a class like `Tuple3` given below produces 10^3 classes, the Cartesian product of 10 variants per type parameter:

```
class Tuple3[A, B, C](a: A, b: B, c:C)
```

In this paper we propose an alternative translation that uses a very simple observation to reduce the bytecode size by orders of magnitude: all primitive types on the Java Virtual Machine can be encoded in the largest value type, the long integer, using a technique we call miniboxing. Storing all value types on the same slot allows a more uniform translation, thus reducing the number of code variants form 10 to 2 per type parameter, corresponding to value and reference types. In the `Tuple3` example, miniboxing only generate 2^3 specialized variants, two orders of magnitude less than specialization. Miniboxed code is faster than homogeneous code, as data access is done directly instead of using pointers and allocating heap memory to store values. Unlike fixnums and tagged unions, miniboxing does not attach the type information to values but to generic classes and methods, leading to less storage used for encoding the types. Furthermore, the full miniboxing transformation eliminates the encoded types carried by classes using load-time cloning and specialization (§6). In this context our paper makes the following contributions:

- Presents an encoding that reduces the specialized variants from 10 to 2 per type parameter, producing orders of magnitude less bytecode (§3) and the transformation which does this (§4)
- Optimizes bulk storage (arrays) in order to reduce the heap footprint and maintain compatibility to homogeneous code (§5)
- Utilizes a load-time class transformation mechanism to avoid megamorphic code and allow standard virtual machine optimizations to kick in (§6)

In order to optimize the code output by the miniboxing transformation, this paper explores the interaction between value encoding and array optimization on the Java Virtual Machine. The insights uncovered here are not tied to the Scala language and may be useful for implementors of other JVM languages too.

The final miniboxing transformation, implemented as a Scala compiler plug-in[1], comes close the performance of monomorphic code and obtains speedups of up to 22x compared to generic code (§7).

2. Specialization in Scala

This section presents specialization [4], a heterogeneous transformation for parametric polymorphism in Scala. Miniboxing builds upon specialization, inheriting its main mechanisms. Therefore a good understanding of specialization and its limitations is necessary to motivate and develop the miniboxing encoding (§3) and transformation (§4).

There are two major approaches to translating parametric polymorphism to Java bytecode: homogeneous, which requires a common representation for all values and heterogeneous, which duplicates and adapts code for each type. By default, both the Scala and Java compilers use homogeneous translation with each value type having a corresponding reference type. Boxing and unboxing operations jump from one representation to the other. For example, `int` has `java.lang.Integer` as its corresponding reference type.

Boxing enables a uniform low level data representation, where all generic type parameters are translated to references. While this simplifies the translation to bytecode, it does come with several disadvantages:

- Initialization cost: allocating an object, initializing it and returning a pointer takes longer than simply writing to a processor register;
- Indirect access: Extracting the value from a boxed type requires computing a memory address and accessing it instead of simply reading a processor register;
- Broken data locality: Seemingly contiguous memory storages, such as arrays of integers, become arrays of pointers to heap objects, which may not necessarily be

[1] `https://github.com/miniboxing/miniboxing-plugin`

aligned in the memory. This can affect cache locality and therefore slow down the execution;

- Heap cost: the boxed object lives on the heap until it is not referenced anymore and is garbage collected. This puts pressure on the heap and triggers garbage collection cycles more often.

To eliminate the overhead of boxing, the Scala compiler features specialization: an annotation-driven, compatible and opportunistic heterogeneous transformation. Specialization is based on the premise that not all code is worth duplicating and adapting: code that rarely gets executed or has little interaction with value types is better suited for homogeneous translation. Since a compile-time transformation such as specialization has no means of knowing how code will be used, it relies on programmers to annotate which code to transform. Recent research in JavaScript interpreters [5, 22] uses profiling as another means to achieve the same goal: compatible specialization of important traces in the program, to eliminate boxing and speed up execution.

With specialization, programmers explicitly annotate the code to be transformed heterogeneously (§2.1 and §2.2) and the rest of the program undergoes homogeneous translation. The bytecode generated by the two translations is compatible and can be freely mixed. This allows specialization to have an opportunistic nature: it injects specialized code, in the form of specialized classes and method calls, (§2.3) but the injected entities are always compatible with the homogeneous translation (§2.4). Although a great design, the interaction with separate compilation leads to certain limitations that miniboxing addresses (§2.5).

2.1 Class specialization

To explain how specialization applies the heterogeneous translation, we can use an immutable linked list example:

```
class ListNode[@specialized T]
       (val head: T, val tail: ListNode[T]) {
  def contains(element: T): Boolean = ...
}
```

Each `ListNode` instance stores an element of type `T` and a reference to the tail of the list. The `null` pointer, placed as the tail of a list, marks its end. A real linked list from the Scala standard library would be more sophisticated [11, 17], but for the purpose of describing specialization this is sufficient. It is also part of the benchmarks presented in the Evaluation section, as it depicts the behavior of non-contiguous collections that require random heap access.

The `ListNode` class has the generic `head` field, which needs to be specialized in order to avoid boxing. To this end, specialization will duplicate the class itself and adapt its fields for each primitive value type. Figure 1 shows the class hierarchy created: the parent class is the homogeneous translation of `ListNode`, which we also call generic class. The 10 subclasses are the specialized variants. They correspond to the 8 Java primitive types, `Unit` (which is Scala's object-

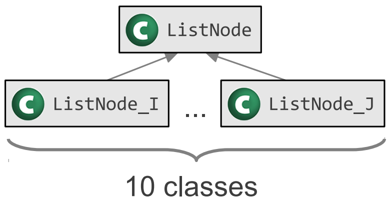

Figure 1. Class hierarchy generated by Specialization. The letters in class suffix represent the type they are specialized for: V-Scala Unit, Z-Boolean, B-Byte ... J-Long, L-AnyRef.

oriented representation of `void`) and reference types[2]. Each of these specialized classes contains a `head` field of a primitive type, and inherits (or overrides) methods defined in the generic class. So far, specialization duplicated the class and adapted the fields, but in order to remove boxing the methods also need to be transformed heterogeneously.

2.2 Method Specialization

In the specialized variants of `ListNode`, the `contains` method needs to be duplicated and adapted to accept primitive values as arguments instead of their boxed representations. Since the `contains` method is already inherited from the generic class, it actually needs to be overridden. But it cannot be overridden, because its signature after the erasure [3] transformation expects a reference type (`java.lang.Object`) and the specialized signature expects a primitive value. Therefore specialized methods need to be name-mangled, giving birth to new methods such as `contains_I` for integers and `contains_J` for long integers.

Nevertheless, the `contains` method from the generic parent class will be inherited in all the specialized classes. However, this is undesirable: all the methods in a specialized class should make use of primitive values. Therefore each specialized class overrides the generic `contains` and redirects it to its mangled variant, such as `contains_I` or `contains_J`. The redirection is done by unboxing the argument received by `contains` and calling the specialized method with the value type, as shown in figure 2.

2.3 Opportunistic Tree Transformation

In the description given so far, specialization does not eliminate boxing yet: in the compiler frontend, the source code is type-checked against generic classes and methods, whether or not they are annotated for specialization. Attempting to instantiate a specialized variant, such as `ListNode_I` or call a specialized method, such as `contains_I` will result in a type checker error. The reason is that the specialization phase runs later in the pipeline, so when type checking occurs, the specialized variants have not been created yet.

The last step in eliminating boxing is rewriting the Scala abstract syntax tree (AST) to instantiate specialized classes and use specialized methods. We call this process rewiring.

[2] Technical note: For a single type parameter the reference variant will not be generated and the generic class will be used instead.

Rewiring works across separate compilation, as the specialized information is written as metadata in the generated bytecode. This makes is possible to use specialized code from libraries.

The instantiation rewiring injects specialized classes when the **new** keyword is used. When the instantiated class has a more specific specialized variant for the given type parameter, the instantiation is rewired. Despite constructing a different class, the types in the AST are not adjusted to reflect this: In the example given below, although the instantiation is rewired to `new ListNode_I`, the type of `node1` remains `ListNode[Int]`. This makes specialization compatible: whether or not the instantiation is rewired, both the specialized class and the generic class are still subtypes of `ListNode[Int]`. Rewiring can only be done if the type parameters are statically known:

```
// before rewiring:
val node1: ListNode[Int] =
      new ListNode[Int](3, null)
// after rewiring:
val node1: ListNode[Int] =
      new ListNode_I(3, null)
// not rewired if U is an abstract type or the
// type parameter of an enclosing class/method
val node2: ListNode[U] =
      new ListNode[U](u, null)
```

The next step of rewiring changes inheritance relations when parent classes have specialized variants that match the type parameters. This injects specialized variants of a class in the inheritance chain, making it possible to use unboxed values when extending a specialized class. This is yet another opportunistic transformation, since the inheritance relation is only rewritten if the type parameter is known statically, as shown by the following example:

```
// before rewiring:
class IntNode(head: Int, tail: IntNode)
       extends ListNode[Int](head, tail)
// after rewiring:
class IntNode(head: Int, tail: IntNode)
       extends ListNode_I(head, tail)
// not rewired, T not known statically:
class MyNode[T](head: T, tail: MyNode[T])
       extends ListNode[T](head, tail)
```

The two rewirings above inject specialized classes in the code. Still, call sites point to the homogeneous methods, which use boxed values. The last rewiring addresses methods, which are rewritten depending on the type of their receiver. Any call site with a specialization-annotated receiver for which the type parameter is statically known is rewritten to use specialized versions of the methods. In the first call site of the example below, the receiver is the specialize-annotated class `ListNode` and the type parameter is statically known to be `Int`. Therefore the call to `contains` is rewired to the specialized `contains_I`:

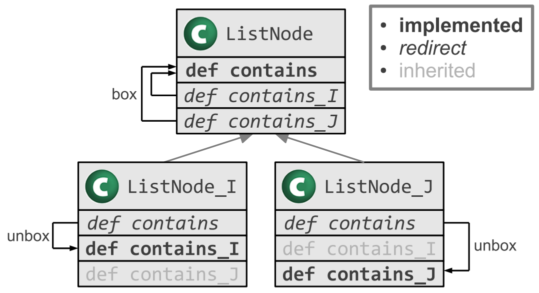

Figure 2. Method overriding and redirection for ListNode and two of its specialized variants.

```
// before rewiring:
(node1: ListNode[Int]).contains(3)
// after rewiring:
(node1: ListNode[Int]).contains_I(3)
// not rewired if U is an abstract type or the
// type parameter of an enclosing class/method
(node2: ListNode[U]).contains(u)
```

2.4 Specialization Compatibility

Since the rewiring process only takes place for statically know type parameters, the generic class and its specialized subclasses may be mixed together. In the following snipped, we see the value `lst` of type `ListNode[Int]` may be either a `ListNode_I` or `ListNode[Int]`:

```
// new ListNode[T] not rewired to
// ListNode_I since T is a type parameter
def node[T](t: T) = new ListNode[T](t, null)

val lst: ListNode[Int] =
  if (random)
    new ListNode[Int](1, null) // ListNode_I
  else
    node(2)                    // ListNode

lst.contains(0) // rewired to contains_I
```

Therefore, calling a specialized method, `contains_I` in this case, can have as receivers both the generic class, `ListNode`, and the specialized one, `ListNode_I`. So both classes must implement the specialized method. To do so, in `ListNode`, `contains` will be implemented using generic code and `contains_I` will box the argument and call `contains`. In `ListNode_I`, `contains_I` will be implemented using primitive value types and `contains` will unbox and redirect. This can be generalized to multiple specialized variants, as can be seen in Figure 2.

This shows the compatible nature of specialization: in order to avoid boxing, both the call site and the receiver need to be rewired, which means the receiver needs to be specialized and the call site needs to know the type parameter statically or be part of code that will be specialized. But if either condition is not fulfilled, the code remains compatible by boxing, either at the call site itself or inside the redirecting method.

2.5 Limitations of Specialization

There are two limitations in specialization: the bytecode explosion and the crippled specialized class inheritance. We will describe each problem and show how the first led to the second.

The specialization mechanism for generating variants is static: whenever the compiler encounters a class annotated for specialization, it generates all its variants up front and includes them in the bytecode output. This is done to support separate compilation.

Theoretically, the specialized variant creation could be delayed until the actual usage but this requires either that the source files for specialized classes are available in all future compilation stages, exactly like in C++, or, if this is undesirable, that the Scala AST is serialized in the generated bytecode, so it can be specialized and compiled later. The former approach is undesirable from a user perspective, as it also requires encoding the original compilation flags and state, while the second is technically very challenging, considering the rich type system and semantics of Scala. Therefore the simplest, although bytecode-expensive solution was chosen: to generate specialized variants for all value types during generic class compilation.

Fulfilling the bytecode compatibility requirements described before, for 10^n variants, means that the generic class needs to implement 10^n methods, all of which are inherited in the subclasses. If they were not inherited, the bytecode would have been proportional to 10^{2n}.

Still, the generic parent design choice affects inheritance between specialized classes. Figure 3 shows an example where the design of specialization bumps into a multiple class inheritance, which is forbidden by Java. In this case, the children inherit from their generic parent, which is suboptimal, since the specialized variants of `MyList` cannot use the specialization in `ListNode`. Experienced Scala programmers might suggest that `MyNode` should be a trait, so it can be mixed in [13]. Indeed this solves the multiple inheritance problem, but creates bytecode proportional to 10^{2n}, because the compiler desugars the trait into an interface, and each specialized `MyList_*` class has to implement the methods in that interface. Other more technical problems stem form this design choice too, but could be avoided by having an abstract parent class. For example, fields from the generic class are inherited by the specialized classes, therefore increasing their memory footprint. Constructors also require more complex code because instantiating a specialized class calls the constructor of its parent, the generic class, which needs to be prevented from running, such that side effecting operations in the constructors are not duplicated.

All in all, at the heart of the bytecode explosion problem and thus the other limitations of specialization, lies the large number of variants per type parameter: 10. For two type parameters, full specialization with correct inheritance creates 10^4 times the bytecode. In practice this is not acceptable. Therefore a natural question to ask is how can we reduce the number of variants generated per type parameter? This is the question that inspired miniboxing.

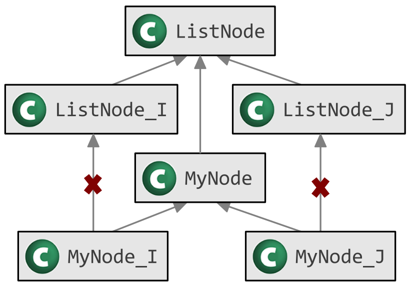

Figure 3. An example of specialized class inheritance made impossible by the current translation scheme

3. Miniboxing Encoding

The bytecode size currently prevents us from extending the use of specialization in the standard library, namely, to tuples of three elements, to the collections hierarchy and to `Function` traits, which are used in Scala's object oriented encoding of functions. Therefore we propose the miniboxing encoding and transformation as a solution to reduce bytecode size and allow library specialization. Along with the encoding, we present a transformation based on the principles of specialization, but using the miniboxed encoding (§4).

Java offers very fast bulk storage in the form of monomorphic arrays. They are often used by collections and algorithms for their fast access and efficient storage. But when used with the miniboxing representation, we need to adapt arrays to support the encoding (§5). This leads to slow programs, which we then optimize using load-time specialization (§6). In the end, we propose a design that makes miniboxed code as fast as monomorphic and specialized code.

The miniboxing encoding uses a very simple insight: all of Java's value types can be encoded on the long integer without losing precision. With the conversion primitives provided by Java, we can treat the long integer like a union type in C, with all primitive values encoded.

Theoretically references could also be encoded using miniboxing, as ultimately they are addresses in the heap. Unfortunately, Java bytecode is not allowed to directly access pointers, as they may be modified by copying garbage collectors. Hence the miniboxing transformation requires two specialized variants per type parameter, one for primitive values and one for references.

Despite reducing the number variants compared to specialization, the size of miniboxed bytecode is still exponential in the number of type parameters. But the size reduces significantly. For example, the same amount of bytecode necessary to fully specialize the `Tuple3` class is generated for `Tuple10` if the miniboxing encoding is used. The evaluation section (§7) provides more comparisons.

The transformation primitives from value types to long and back are implemented in the Java virtual machine and have direct translations to bytecode and to processor instructions [1]. Nevertheless, several concerns need our attention when using miniboxing:

- Packing and unpacking cost
- Memory footprint of the miniboxed encoding
- Executing operations on miniboxed values (§4)

Packing and unpacking cost. Boxing and unboxing accesses the heap memory. The main goal of miniboxing is to eliminate this overhead, but, in doing so, conversions to and from long integers must not slow down program execution significantly compared to monomorphic code. Our benchmarks show that indeed the overhead is negligible (§7).

Memory footprint. The miniboxed encoding has a memory footprint between that of monomorphic and generic code. Considering byte as the type parameter, the memory footprint of the miniboxed encoding is 8 times larger than the one for monomorphic code, which would store the byte directly. This factor is reduced by specializing bulk storage (§5) and considering the object headers and paddings introduced by the Java Virtual Machine. On the other hand, when comparing to generic code running on 64 bit processors, the factor is exactly 1, as both a pointer and a long integer have 8 bytes. And this does not take into account the heap space occupied by the boxed values themselves. All things considered, miniboxing is not a technique to reduce memory footprint but rather to eliminate the overhead of boxing, and doing so without bytecode explosion.

The last point in the list, executing operations on miniboxed values, is addressed in the next section, which further describes the miniboxing code transformation.

4. Miniboxing Transformation

The miniboxing transformation, which we developed as a Scala compiler plugin, builds upon specialization. It has the same opportunistic and compatible nature of specialization and performs class and method duplication in a similar manner. Still, four elements set it apart:

- the use of parent traits to allow miniboxed class inheritance
- the use of the miniboxing encoding to store values and type bytes to store the data type
- the runtime support for miniboxed values (§5)
- the load-time specialization mechanism (§6)

Miniboxing uses a generic trait as the parent of the specialized classes, therefore avoiding the limitation that miniboxed classes cannot inherit from each other (§2.5). Figure 4 shows an example miniboxed class inheritance. Having a trait as the parent increases the bytecode size from 2^n to 4^n, as each of the 2^n miniboxed variants need to implement all 2^n methods.

Operations on miniboxed values. The miniboxed encoding does not carry type information along with values. Instead, type information is attached to miniboxed classes: for each type parameter that is encoded using miniboxing, let's call it `T`, the transformation adds an extra field `T_Type` to the class and its constructor. This extra type field encodes the primitive type that was bound to the type parameter. Now, for all values encoded on a long integer that had type `T` before miniboxing, the transformation uses `T_Type` to dispatch the correct operations, such as `toString`, `hashCode`, `equals`, boxing conversions and array operations. Since classes usually manipulate several values of the same generic type, encoding types in the class instead of the values saves memory and simplifies the encoding.

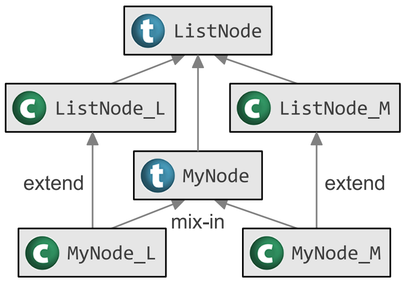

Figure 4. An example of miniboxed class inheritance. The suffixes are: M - miniboxed encoding and L - reference type

Type fields also need to be added to minibox-duplicated methods which receive or return miniboxed values. The reason is that classes which encode references do not receive the global type field, but they do need to encode or decode miniboxed values. These arguments are ignored in the variants which use the miniboxed encoding in favor of the class-wide type field. This can be illustrated for the reference-encoding variant of `ListNode` in the miniboxing translation:

```
// ListNode miniboxed variant for references:
class ListNode_L[T](val head: T, val tail:
    ListNode[T]) {
  def contains(element: T): Boolean = ...
  def contains_M(element: Long, T_Type_m: Byte):
      Boolean =
     contains(mb2box(element, T_Type_m))
}
```

The type field only encodes the 9 primitive types in Scala, therefore it does not incur the typical overhead of reified types [19]. We have explored two implementations for type fields: bytes that can be used in switch statements, which we call type bytes (§5.3) and type descriptor objects, which we call dispatchers (§5.4).

Operations on arrays also use the type field: The miniboxing transformation rewires all array methods to special runtime support calls, which along with the array, take a type field representing the array's element type, as can be observed in the `ArrayBuffer` class, in the next example:

```
// ArrayBuffer miniboxed variant for primitives:
class ArrayBuffer_M[T: Manifest](T_Type: Byte)
                              extends ArrayBuffer[T] {
  // fields, getters and setters:
  private[this] var _array_M: Array[T] =
      mbarray_new(32, T_Type)
  def array: Array[T] = array_M
  def array_M: Array[T] = _array_M
  ...

  // methods:
  private def getElement(idx: Int): T =
     mb2box(getElement_M(idx, T_Type), T_Type)
  private def getElement_M(idx: Int, T_Type_m:
      Byte): Long =
     mbarray_get(array_M, idx, T_Type)
  ...
}

// rewritten to new ArrayBuffer_M[Int](INT):
val a = new ArrayBuffer[Int]()
```

The runtime support, which is called via `mbarray_new`, `mbarray_get` and `mb2box` in the example above, is the most crucial piece of the miniboxing transformation: slight changes in how the runtime support is implemented can have significant impact on the overall performance of miniboxed code. Therefore the runtime support methods have to be small enough to be inlined by the virtual machine and to be structured in ways that return the result as fast as possible for any of the primitive types. The following two sections describe the runtime support for arrays and give technical insights into the pitfalls of the implementation.

5. Miniboxing Bulk Storage Optimization

Arrays are Java's bulk storage facility. They can store value types or references to heap objects. This is done efficiently, as values are stored one after the other in contiguous blocks of memory and access is done in constant time. Their characteristics make arrays good candidates for internal data structures in collections and algorithms.

But in order to implement compact storage and constant access overhead, arrays are monomorphic under the hood, with separate (and incompatible) variants for each of the primitive value types. What's more, each array type has its own specific bytecode instructions to manipulate it.

The goal we set forth was to match the performance of monomorphic arrays in the context of miniboxing-encoded values. To this end, we have two alternatives to implementing arrays for miniboxed values: use arrays of long integers to store the encoded values or use monomorphic arrays for each type, and encode or decode values at each array access.

Storing encoded values in arrays provides the advantage of uniformity: all the code in the minibox-specialized class uses the long representation and array access is done in a single instruction. Although this representation wastes heap space, especially for small value types such as boolean or byte, this is not the main drawback: It is incompatible with the rest of the Scala code.

Scala code uses monomorphic arrays for each value type, therefore long integer arrays in miniboxed classes must not be allowed to escape from the transformed class, otherwise they may crash the code attempting to read or write them. To improve compatibility, we could convert escaping arrays to their specialized forms. This would not only introduce delays, but also provide an incorrect solution, as writes from the Scala code would not be visible in the miniboxed code and vice versa. After analyzing the solutions we decided against using encoded values in arrays.

Thus, we chose to use arrays in their monomorphic format for each value type, so we maintain compatibility with the rest of the Scala code. This decision brought with it another problem: any array access requires a call to the miniboxing runtime support which performs a dispatch on the type byte. Depending on the type byte's value, the array is cast to its correct type and the corresponding bytecode for accessing it is invoked. This is followed by the encoding operation, which converts the read value to a long integer. The following snippet shows the array read operation implemented in the miniboxing runtime support code:

```
def mbarray_apply_minibox[T](array: Array[T],
    idx: Int, tag: Byte): Minibox = tag match {
  case INT =>
    array.asInstanceOf[Array[Int]](idx).toLong
  case LONG =>
    array.asInstanceOf[Array[Long]](idx)
  case DOUBLE => Double.doubleToRawLongBits(
    array.asInstanceOf[Array[Double]](idx)).toLong
  ...
}
```

The most complicated and time-consuming part of our work focused on matching the performance of monomorphic code using the miniboxing runtime support. The next subsections present the Java Virtual Machine inlining process (§5.1), the benchmark we used for testing (§5.2) and the two implementations for runtime support: type byte switching (§5.3) and object-oriented dispatching (§5.4).

5.1 Java Virtual Machine Inlining

We used benchmarks to guide our implementation of the miniboxing runtime support on the HotSpot Java Virtual Machine. The Evaluation section (§7) provides full details on the benchmarking procedure. Nevertheless, we will briefly present the optimization and compiling mechanisms [9, 15] in the Java Virtual Machine because they drove our design decisions. Although the work was done based on the Java Virtual Machine, we highlight the underlying mechanisms that transform our code and derive a set of guidelines that should be applicable to other virtual machines as well, as long as they also perform interpretation and compilation with deoptimization.

The Java Virtual Machine starts off by interpreting bytecode. After several executions, a method is considered hot and the compiler is called in to transform it into native code that can be executed faster. During compilation, aggressive inlining is done recursively on all the methods that have been both executed enough times and are small enough. Typical numbers are 10000 executions for the C2 (server) compiler and 35 bytes for inlining.

When inlining static calls, the code is available and is inlined directly. For virtual and interface calls, the types are profiled during interpretation and the compiler inlines the code from the class known to be the runtime receiver. The inlining performed may later become incorrect, if a different class is used as the receiver. For such a case the compiler inserts a guard: if the runtime type is not the one expected, it bails out to interpretation mode. The bytecode may be compiled again later if it runs enough times. But if a third runtime receiver is seen, the call site is marked as megamorphic and no inlining is performed anymore.

In the object oriented dispatchers implementation we found that slowdowns were occurring because of the changing nature of miniboxed code, which takes very different paths through the code depending on the encoded value type, thus forcing the Java Virtual Machine to back off its optimizations as new paths become common and older paths become unused.

After inlining as much code as feasible, the compiler has the code applies optimizations, which can significantly reduce running time, especially for array operations which are very regular.

5.2 Benchmark

We benchmarked the performance on the two examples presented throughout the paper, `ListNode` and `ArrayBuffer` and noticed one particular method stood out as the most sensitive to runtime support implementation: the `reverse` method of the `ArrayBuffer` class. Therefore in this section we use it to show how well each version of the miniboxing runtime support performs:

```
def reverse_M(): Unit = {
  var idx = 0
  val xdi = elemCount_M - 1
  while (idx < xdi) {
    val el1: Long = getElement_M(idx, T_Type)
    val el2: Long = getElement_M(xdi, T_Type)
    setElement_M(idx, el2, T_Type)
    setElement_M(xdi, el1, T_Type)
    idx += 1
    xdi -= 1
  }
}
```

The running times presented in table 1 correspond to reversing an integer array buffer of 3 million elements. To put things into perspective, along with different designs, the table also provides running times for monomorphic (code specialized by hand), specialization-annotated and generic code. Measurements are taken in two scenarios: For "Single Context", an array buffer of integers is created and populated and its `reverse` method is benchmarked. In "Multi Context", the array buffer is instantiated, populated and reversed for all primitive value types first. Then a new array buffer of integers is created, populated and its `reverse` method benchmarked. The Java Virtual Machine optimizations are influenced by the historical paths executed in the program, so using other type parameters can have a drastic impact on performance, as can be seen from the table, where the times for "Single Context" and "Multi Context" are different: this means the Java Virtual Machine backed off its optimizations when the benchmark was executed with different types.

	Single Context	Multi Context
generic	20.4 ± 3.7	21.5 ± 2.2
miniboxed, no inlining	34.5 ± 3.1	34.4 ± 2.2
miniboxed, full switch	2.4 ± 0.6	15.1 ± 3.5
miniboxed, semi-switch	2.4 ± 0.4	17.2 ± 3.0
miniboxed, decision tree	24.2 ± 3.0	24.1 ± 2.9
miniboxed, linear	24.3 ± 3.0	22.9 ± 4.0
miniboxed, dispatcher	2.1 ± 0.6	26.4 ± 1.9
specialized	2.0 ± 0.6	2.4 ± 0.4
monomorphic	2.1 ± 0.6	N/A

Table 1. The time in milliseconds necessary for reversing an array buffer of 3 million integers. Performance varies based on how many value types have been used before (Single Context vs. Multi Context).

5.3 Type Byte Switching

The first approach we tried, the simple switch on the type byte, quickly revealed a problem: The array runtime support methods were too large for the virtual machine to inline at runtime. This corresponds to the "miniboxing, no inlining" in table 1. To solve this problem, we tasked the Scala compiler with inlining runtime support methods in its backend, independently from the virtual machine. But this was not enough: the `reverse_M` method calls `getElement_M` and `setElement_M`, which also became large after inlining the runtime support, and were not inlined by the virtual machine. This required us to recursively mark methods for inlining between the runtime support and the final benchmarked method.

The forceful inlining in the Scala backend produced good results. The measurement, corresponding to the "miniboxed, full switch" row in the table, shows miniboxed code working at almost the same speed as specialized and monomorphic code. This can be explained by the loop unswitching optimization in the virtual machine compiler. With all the code inlined by the Scala backend, loop unswitching was able to hoist the type byte switch statement outside the reverse loop. It then duplicated the loop contents for each case in the switch. Inside the duplicated loop, array-specific optimizations were able to reduce the running time down to almost the same as monomorphic code.

But using different primitive types diminished the benefit. We tested the reverse operation in two situations, to check if the optimizations still kick in after we run the same method for different type parameters. It is frequently the case that the

Java Virtual Machine will compile a method with aggressive assumptions about which paths the execution may take. For the branches that are not taken, guards are left in place. Then, if a guard is violated during execution, the native code is interrupted and the program continues in the interpreter. The method may be compiled again later, if it is executed enough times to warrant compilation to native code. Still, upon recompilation, the path that was initially compiled to a stub now becomes a legitimate path and may preclude some optimizations. We traced this problem to the conversion between floating point numbers and integers, which has the special status of intrinsic method and prevents the loop unswitching.[3]

We tried different constructions for the miniboxing runtime support: splitting the match into two parts and having an if expression that would select one or the other ("semi-switch"), transforming the switch into a decision tree ("decision tree") and using a linear set of 9 if statements ("linear"), all of which appear in table 1. These new designs either degraded in the multiple context scenario, or provided a bad baseline performance from the beginning. It was clear there was a need for a different solution.

5.4 Dispatching

The results obtained with type byte switching showed that we were committing to a type too late in the execution: Forced inlining had to carry our large methods that covered all types inside the benchmarked method, where the virtual machine had to hoist the switch outside the loop:

```
while (...) {
  val el1: Long = T_Type match { ... }
  val el2: Long = T_Type match { ... }
  T_Type match { ... }
  T_Type match { ... }
}
```

This is wrong. The type decision should be taken as early as possible, even as soon as object creation. This can be done using an object-oriented approach: instead of passing a byte value during class instantiation and later switching on it, we can pass objects which encode the runtime operations for a single type. We call this object the dispatcher. The dispatcher for each value type encodes a common set of operations such as array get and set.

```
abstract class Dispatcher {
  def array_get[T](arr: Array[T], idx: Int)
  def array_update[T](arr: Array[T], idx: Int,
      elt: Long)
  ...
}
object IntDispatcher extends Dispatcher { ... }
```

Dispatcher objects are passed to the miniboxed class during instantiation and have final semantics. In the `reverse` benchmark, this would replace the type byte switches by method invocations, which could be inlined. Dispatchers make the forceful inlining and loop unswitching redundant. With the final `dispatcher` field set at construction time, the `reverse_M` inner loop body can have array access inlined and optimized: ("miniboxed-dispatchers" in table 1)

```
// inlined getElement:
val el1: Long = dispatcher.array_get(...)
val el2: Long = dispatcher.array_get(...)
// inlined setElement:
dispatcher.array_update(...)
dispatcher.array_update(...)
```

Despite their clear advantages, in practice dispatchers can be used with at most two different value types. This happens because the Java virtual machine inlines the dispatcher code at the call site and installs guards that check the object's runtime type. The inline cache works for two receivers, but if we try to swap the dispatcher a third time, the callsite becomes megamorphic. In the megamorphic state, the code will not be inlined, hence the disappointing results for the "Multi Context" scenario.

Interestingly, specialization performs equally well in both "Single Context" and "Multi Context" scenarios. The explanation lies in the bytecode duplication: each specialized class contains a different body for the reverse method, and the classes for different value types do not interact. Accordingly, the results for integers are not influenced by the other value types used. The next section presents load-time cloning and specialization.

6. Miniboxing Runtime Optimization

The miniboxing runtime support, in both incarnations, type byte and dispatcher, fails to deliver performance in the "Multi Context" scenario. The reason, in both cases, is that execution takes multiple paths through the code and this prevents the Java virtual machine from optimizing. Therefore an obvious solution is to duplicate the class bytecode, but instead of duplicating it on the disk, we can do it in memory, at runtime. The .NET common language runtime performs specialization [8] at runtime, but it does so using more complex transformations encoded in the virtual machine.

We use a custom classloader to clone and specialize miniboxed classes. Similar to the approach in *Pizza* [14], the classloader takes the name of a class, which embeds the type byte value. For example, `ListNode_5` corresponds to a clone of `ListNode_M` with the type byte set to 5. From the name, the classloader infers the miniboxed class name and loads it from the classpath. It clones the bytecode and adjusts the owner in the constant table. All this is done in-memory and on-demand.

Once a class' bytecode is cloned, the paths taken through the runtime support in each class remain fixed during its lifetime, making the performance in "Single Context" and "Multi Context" comparable. The explanation is that the

[3] We are not aware of the exact reason this happens, but it may be related to conservative assumptions about intrinsic code.

JVM sees different classes, with separate type profiles, for each primitive type.

Aside from bytecode cloning, the classloader also performs class specialization:

- Avoids the extra type tag fields in cloned class, by making them static (as the class is already dedicated to a type)
- Uses constant propagation and dead code elimination to reduce each type tag switch down to a single case, which can be inlined by the virtual machine, thus eliminating the need for forced inlining
- Performs runtime rewiring to cloned classes, which is described in the next section

6.1 Miniboxing Runtime Rewiring

The miniboxing transformation follows the same rules set forth by specialization for rewiring. But runtime cloning introduces a new layer of rewiring, which needs to take the cloned classes into account. The factory mechanism we employ (§6.2) is equivalent to to the instance creation rewiring done by specialization. The two other rewiring mechanisms employed by specialization are method rewiring and parent class rewiring. Fortunately method rewiring is done statically throughout the user code and since methods are not modified, there is no need to rewire them in the classloader. Still, the classloader must to rewire parent classes.

Parent rewiring allows a specialized class to inherit and use specialized methods from its parent. If the parent specialization is done in a purely static fashion, all classes extending `ArrayBuffer_M` share the same code for the `reverse_M` method. Therefore using them with different value types brings back the "Multi Context" problem: using multiple value types degrade performance. Rewiring parent classes is done both at compile time, to the miniboxed variant of the class, and at runtime by the classloader, to the correct cloned class. The following snippet gives an example of rewiring:

```
// user code:
class IntBuff extends ArrayBuffer[Int]
// after static rewriting:
class IntBuff extends
    ArrayBuffer_M[Int](IntDispatcher)
// after runtime rewriting:
class IntBuff extends
    ArrayBuffer_I[Int](IntDispatcher)
```

The rewiring of runtime parent classes requires that all children which extend a minibox-specialized class go through the factory instantiation, so their parents can be modified by the classloader. This includes the classes extending miniboxed parents with static type parameters, such as for the `IntBuff` class in the code snippet before. This is a major inconvenience especially for classes that are only used once, such as anonymous closures extending `FunctionX`, because the first factory instantiation has a non-negligible overhead which doesn't pay off. But such classes do not make use of the miniboxing runtime for arrays, therefore

	Single Context	Multi Context
generic	20.4 ± 3.7	21.5 ± 2.2
miniboxed, full switch	2.4 ± 0.6	15.1 ± 3.5
mb. full switch, RT	2.5 ± 0.5	**2.4** ± 0.5
miniboxed, dispatcher	2.1 ± 0.6	26.4 ± 1.9
mb. dispatcher, RT	2.0 ± 0.5	**2.7** ± 0.1
specialized	2.0 ± 0.6	2.4 ± 0.4
monomorphic	2.1 ± 0.6	N/A

Table 2. The time in milliseconds necessary for reversing an array buffer of 3 million integers. Miniboxing benchmarks were ran with runtime cloning mechanism. (RT)

don't need runtime cloning at all. To avoid useless cloning, we can devise an annotation which hints the compiler which classes need factory instantiation. This annotation can be automatically added by the compiler when a class uses array operations and propagated from parent classes to their children:

```
@runtimeDup
class ArrayBuffer[@minispec T]

// IntBuff automatically inherits @runtimeDup
class IntBuff extends ArrayBuffer[Int]
```

6.2 Efficient Instantiation

Imposing the use of a global classloader is impossible in many practical applications. To make miniboxing usable in practice, we chose to perform the class instantiation through a factory that loads a local specializing classloader, requests the cloning and specialization of a miniboxed class and instantiates it via reflection. We benchmarked the approach and it introduced significant overhead, as instantiations using reflection are very expensive.

To counter the cost of reflective instantiation, we propose a "double factory" approach that uses a single reflective instantiation per cloned class. In this approach each cloned and specialized class has a corresponding factory – which instantiates it using the new keyword. When instantiating a miniboxed class with a new set of parameters, its corresponding factory is specialized by the classloader and instantiated by reflection. From that point on, any new instance is created by the factory, without the reflective delay. The following code snippet shows the factory instantiation:

```
// Factory interface
abstract class ArrayBufferFactoryInterface {
  def newArrayBuffer_M[T: Manifest](disp:
      Dispatcher[T]): ArrayBuffer[T]
}
// Factory instance, to be specialized
// in the classloader
class ArrayBufferFactoryInstance_M extends
    ArrayBufferFactoryInterface {
  def newArrayBuffer_M[T: Manifest](disp:
      Dispatcher[T]): ArrayBuffer[T] =
    new ArrayBuffer_M(disp)
}
```

7. Evaluation

This section presents the results obtained by the miniboxing transformation. It will first present the miniboxing Scala compiler plug-in and the miniboxing classloader. Next, it will present the benchmarking infrastructure and the benchmark targets. Finally, it will present the results and draw conclusions.

7.1 Miniboxing Plug-in and Classloader

The miniboxing plug-in adds a code transformation phase in the Scala compiler. Like specialization, the miniboxing phase is composed of two steps: transforming signatures and transforming trees. As the signatures are specialized, metadata is stored on exactly how the trees need to be transformed. This metadata later guides the tree transformation in duplicating and adapting the trees to obtain the miniboxed code. The duplication and adaption step reuses the infrastructure from specialization, with the necessary updates to support the more complex transformations required by miniboxing.

The tree transformation contains several steps:

- code duplication and adaptation, where values of type `T` are replaced by long integers and are un-miniboxed back to `T` at use sites
- runtime support rewiring for methods like `toString`, `hashCode`, `equals` and array operations
- opportunistic rewiring: new instance creation, specialized parent classes and method invocations
- peephole minibox/un-minibox canceling

The miniboxing plug-in currently transforms most simple Scala classes. Since our main goal so far has been to match the speed of monomorphic code, we deliberately avoided complexity. For example, our plug-in is not capable of performing partial transformation, where only some of the type parameters participate in the heterogeneous translation. Also we do not support internal classes and type parameters with bounds. Finally, the plug-in does not support method specialization. Nevertheless, these more advanced features are available in specialization, and we can adapt and reuse them in future releases.

The miniboxing classloader duplicates classes and performs new instance rewiring. It uses transformations from an experimental Scala backend to perform constant propagation and dead code elimination that remove type byte switches. It supports miniboxed classes generated by the current plug-in. The infrastructure for the double factory instantiation was written and tuned by hand, and will be integrated in the plug-in in the next release. We also did not implement the `@runtimeDup` annotation yet.

The project also contains code for testing the plug-in and the classloader and performing microbenchmarks, something which turned out to be more difficult than expected.

7.2 Benchmarking Infrastructure

The miniboxing plug-in produces bytecode which is executed by the Java Virtual Machine. Although the virtual machine provides useful services to the running program, such as compilation, deoptimization and garbage collection, these operations influence our microbenchmarks by delaying or even changing the benchmarked code altogether. Furthermore, the non-deterministic nature of such events make proper benchmarking harder [6].

In order to have reliable results for our microbenchmarks, we used ScalaMeter [16], a tool specifically designed to reduce microbenchmarks noise. ScalaMeter is currently used in regression testing of the Scala standard library. When benchmarking, it forks a new virtual machine such that fresh code caches and type profiles are created. It then warms up the benchmarked code until the virtual machine compiles it down to native code using the C2 compiler. When the code has been compiled and the benchmark reaches a steady state, ScalaMeter measures several execution runs. The process is repeated several times, 100 in our case, making the benchmarks less noisy. For the report, we average the measurements and present the standard deviation.

We execute our benchmarks on an 8-core i7 machine running at 3.40GHz with 16GB of RAM memory. The machine ran a 64 bit version of Linux Ubuntu 12.04.2. For the Java Virtual Machine we used the Oracle Java SE Runtime Environment build 1.7.0_11 using the C2 (server) compiler. The following section will describe the benchmarks we ran.

7.3 Benchmarks

We executed the benchmarks in two scenarios:

- “Single Context” corresponds to the benchmark target (`ArrayBuffer` or `List`) executed with a single value type, `Int`
- “Multi Context” corresponds to running the benchmark for all value types and only then measuring the execution time for the target value type, `Int`

The benchmarks were executed with 7 transformations:

- generic - the generic version of the code, uses boxing
- mb. switch - miniboxed, using the type byte switching
- mb. dispatcher - miniboxed, dispatcher runtime support
- mb. switch + RT - miniboxed, type byte switching runtime support, classloader duplication
- mb. dispatcher + RT - miniboxed, dispatcher, classloader
- specialized - code transformed by specialization
- monomorphic - code specialized by hand, which does not need the redirects generated by specialization

For the benchmarks, we used the two classes presented in the previous sections: The `ArrayBuffer` class, that simulates collections and algorithms which make heavy use of bulk storage and the `ListNode` class, that simulates collections which require random heap access. We chose the benchmark methods such that each tested a certain feature of

	`ArrayBuffer.append`		`ArrayBuffer.reverse`		`ArrayBuffer.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	50.1 ± 1.2	48.0 ± 0.9	20.4 ± 3.8	21.5 ± 2.3	1580.1 ± 12.0	3628.8 ± 40.2
mb. switch	30.9 ± 2.0	35.5 ± 2.4	2.5 ± 0.5	15.1 ± 3.5	161.5 ± 3.4	554.3 ± 3.2
mb. dispatch	16.5 ± 1.3	58.2 ± 3.0	2.1 ± 0.6	26.5 ± 1.9	160.7 ± 3.0	2551.6 ± 33.1
mb. switch + RT	**15.6** ± 1.8	**14.8** ± 1.5	**2.5** ± 0.5	**2.4** ± 0.5	**159.9** ± 3.2	**161.7** ± 3.4
mb. dispatch + RT	15.1 ± 0.9	15.9 ± 1.5	2.0 ± 0.6	2.7 ± 0.1	161.8 ± 2.2	161.3 ± 2.8
specialization	39.7 ± 3.0	38.5 ± 3.0	2.0 ± 0.6	2.4 ± 0.5	155.8 ± 2.6	156.3 ± 3.4
monomorphic	16.2 ± 1.0	N/A	2.1 ± 0.6	N/A	157.7 ± 4.1	N/A
	`List` creation		`List.hashCode`		`List.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	16.7 ± 0.2	1841 ± 1068	22.1 ± 1.9	20.4 ± 0.8	1739.5 ± 4.9	2472.4 ± 49.0
mb. switch	11.4 ± 0.1	11.7 ± 0.6	18.3 ± 2.2	18.8 ± 2.2	1438.2 ± 17.5	1443.2 ± 21.8
mb. dispatch	11.4 ± 0.1	11.5 ± 0.4	15.6 ± 1.8	21.0 ± 1.9	1369.1 ± 8.0	1753.2 ± 16.9
mb. switch + RT	**11.5** ± 0.6	**11.6** ± 0.5	**16.2** ± 2.4	**16.1** ± 2.2	**1434.9** ± 20.4	**1446.3** ± 19.6
mb. dispatch + RT	12.1 ± 0.5	12.7 ± 1.9	16.1 ± 2.1	15.3 ± 1.7	1364.2 ± 7.8	1325.9 ± 44.4
specialzation	11.4 ± 0.1	11.4 ± 0.2	14.5 ± 1.4	36.4 ± 2.5	1341.0 ± 37.8	1359.2 ± 23.4
monomorphic	10.2 ± 1.2	N/A	13.3 ± 0.8	N/A	1172.0 ± 3.4	N/A

Table 3. Benchmark running times and standard deviation. JVM compilation and optimization. The time is in milliseconds.

the miniboxing transformation. We used very small methods such that any slowdowns can easily be attributed to bytecode or can be diagnosed in a debug build of the virtual machine, using the compilation and deoptimization outputs.

`ArrayBuffer.append` creates a new ArrayBuffer and appends 3 million elements to it. This benchmark tests the array writing operations in isolation, such that they cannot be grouped together and optimized.

`ArrayBuffer.reverse` reverses a 3 million element array buffer. This benchmark proved the most difficult in terms of matching the monomorphic code performance, as it was heavily optimized by the virtual machine.

`ArrayBuffer.contains` checks for the existence of elements inside an initialized array buffer. It exercises the `equals` method rewiring and hinted us that the initial transformation for `equals` was suboptimal, as we were not using the information that two miniboxed values were of the same type. This is the benchmark that showed a 22x speedup over generic code.

List construction builds a 3 million element linked list using `ListNode` classes. This benchmark verifies the speed of miniboxed class instantiation. It was heavily slowed down by the single factory approach, therefore we introduced the double factory for class instantiation using the classloader.

`List.hashCode` computes the hash code of a list of 3 million elements. We used this benchmark to check the performance of the `hashCode` rewiring. It was a surprise to see the `hashCode` performance for generic code running in the interpreter (table 4). It is almost one order of magnitude faster than specialized code and 5 times faster than miniboxing. The explanation is that computing the hashCode requires boxing and calling the hashCode method on the boxed object. When the benchmarks are compiled and optimized, this is avoided by inlining and escape analysis, but in the interpreter, the actual object allocation and call to `hashCode` do happen, making any type of heterogeneous translation slower.

	`ArrayBuffer`			`List`		
generic	4.6	2.2	367.0	1.4	**0.2**	16.6
mb. dispatch + RT	2.5	0.7	88.9	1.1	1.5	7.3
mb. switch + RT	**1.6**	**0.3**	**25.0**	**0.8**	**1.3**	**4.2**
specialization	4.3	0.5	30.7	0.6	1.9	2.2
monomorphic	1.0	0.2	12.7	0.4	1.2	2.2

Table 4. Running time for the benchmarks in the Java Virtual Machine interpreter. The time is in seconds.

`List.contains` tests whether a list contains an element, repeated for 3 million elements. It tests random heap access and the performance of the `equals` operator rewiring.

7.4 Interpreter Benchmarks

In table 4 we present the same set of benchmarks ran in the Java Virtual Machine interpreter. It is important that transformations do not significantly degrade performance in the interpreter, as this slows down application startup. It also affects `for` loops where the closure is executed only a few times.

7.5 Bytecode Size

Table 5 presents the bytecode generated by 3 transformations: the miniboxing with dispatcher, miniboxing with switching and specialization. The fraction of bytecode created by miniboxing lies between 0.2x to 0.4x. This is marginally better than the fraction we expected, 0.4x, which corresponds to $4^n/10^n$ for $n = 1$. Still, the double factory mechanism generates a significant amount of bytecode, around 10 kilobytes per class. In the future releases we hope to eliminate this overhead at least to some extent.

	dispatcher	switching	specialized
ArrayBuffer	19.5	24.5	57.6
ArrayBuffer factory	9.0	8.5	–
ListNode	10.9	11.5	45.0
ListNode factory	8.7	8.3	–

Table 5. Bytecode generated by different translations, in kilobytes. Factory represents the double factory mechanism for the runtime cloning and specialization.

7.6 Evaluation Remarks

After analyzing the benchmarking results, we believe the miniboxing transformation with type byte switching and classloader duplication provides the most stable results and fulfills our initial goal of providing an alternative encoding for specialization, which produces less bytecode without sacrificing performance. Using the classloader for duplication and switch elimination, the type byte switching does not require forced inlining anymore, making the transformation work without backend annotations.

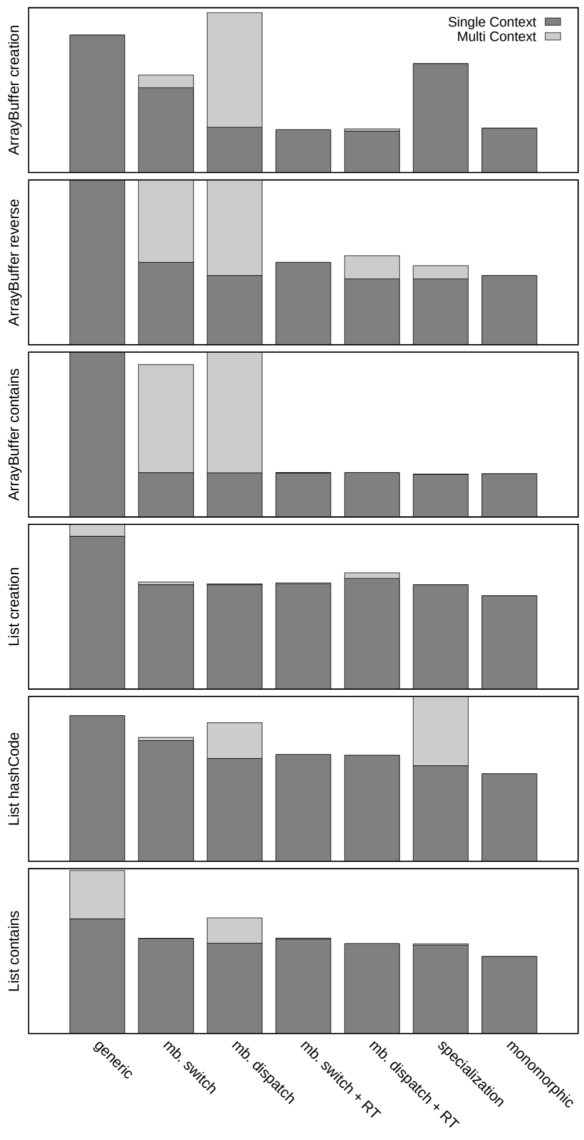

Figure 5. Data in table 3 in graphical form.

8. Related Work

The work by *Sallenave* and *Ducournau* [18] shares the same goals as miniboxing: offering unboxed generics without the bytecode explosion. However, the target is different: their Lightweight Generics compiler targets embedded devices and works under a closed world assumption. This allows the compiler to statically analyze the .NET bytecode and conservatively approximate which generic classes will be instantiated at runtime and the type parameters that will be used. This information is used to statically instantiate only the specialized variants that may be used by the program. To further reduce the bytecode size, instantiations are aggregated together into three base representations: `ref`, `word` and `dword`. This significantly reduces the bytecode size and does not require runtime specialization. To dispatch operations, the Lightweight Generics compiler uses the reified types available at runtime [8]. At the opposite side of the spectrum, miniboxing works under an open-world assumption, and inherits the opportunistic and compatible nature from specialization, which enables it to work under erasure [3], without the need for runtime type information. Instead, type bytes are a lightweight and simple mechanism to dispatch operations for encoded value types. Finally, the runtime cloning and specialization mechanism eliminates the cost of dispatching operations, something which is not available on limited embedded devices.

According to *Morrison et al* [12] there are three types of polymorphism: *textual polymorphism*, which corresponds to the heterogeneous translation, *uniform polymorphism* which corresponds to the homogeneous translation and *tagged polymorphism* which creates uniform machine code that can handle non-uniform store representations. In the compiler they develop, for the *Napier88* language, the generated code uses a tagged polymorphism approach with out-of-band signaling, meaning the type information is not encoded in the values themselves. Their encoding scheme accommodates surprisingly diverse values: primitives, data structures and abstract types. As opposed to the Napier88 compiler, the miniboxing transformation is restricted to primitives. Nevertheless, it can optimize more using the runtime specialization approach, which eliminates the overhead of tagging. Furthermore, the miniboxing runtime support allows the Java Virtual Machine to aggressively optimize array operations, which makes bulk storage operations orders of magnitude faster. The initial runtime support implementations presented in section 5 show that it is not possible to have these optimizations in a purely compiler-level approach, at least not on the current incarnation of the Java Virtual Machine.

Fixnums in Lisp [21] reserve bits for encoding the type. For example, an implementation may use a 32-bit slot to encode both the type, on the first 5 bits and the value, on the last 27 bits. We call this in-band type signaling, as the type is encoded in the same slot as the payload. Although very efficient in terms of space, the in-band encoding has two drawbacks that we avoid in the miniboxing encoding: the ranges of integers and floating point numbers are restricted to only 27 bits and each operation needs to unpack the type, dispatch the correct routine and pack the value with its type back. This requires a non-negligible amount of work for each operation. Out-of-band types are also used in Lua [7], where they are implemented using tagged unions in C. The difference that sets miniboxing apart consists in attaching the out-of-band type information to classes and methods instead of values, therefore reusing the encoded type information and removing unnecessary storage. This separate encoding is made possible by the static guarantees the type system offers, such as, for example, that two parameters to a method are of the same type.

The .NET common language runtime [8] was a great inspiration for the specializing classloader. It stores generic templates in the bytecode, and instantiates them in the virtual machine for each type parameter used. Two features are crucial in enabling this: the global presence of reified types and the instantiation machinery in the virtual machine. Contrarily, the Java Virtual Machine does not store representations of the type parameters at runtime [3] and re-introducing them globally is very costly [19]. Therefore, miniboxing needs to inherit the opportunistic behavior from specialization. On the other hand, the classloading mechanism for template instantiation at runtime is very basic, and not really suited to our needs: it is both slow, since it uses reflection, and does not allow us to modify code that is already loaded from the classpath. Consequently we were forced to impose the double factory mechanism for all classes that extend or mix-in miniboxed parents, creating redundant boilerplate code, imposing a one-time overhead for class instantiation and increasing the heap requirements.

The *Pizza* generics support [14] inspired us in the use of traits as the base of the specialized hierarchy, also offering insights into how class loading can be used to specialize code. The mechanism employed by the classloader to support arrays is based on annotations, which mark the bytecode instructions that need to be patched to allow reading an array in conformance with its runtime type. In our case there is no need for patching the bytecode instructions, as miniboxing goes the other way around: it includes all the code variants in the class and then performs a simple constant propagation and dead code elimination to only keep the right instruction. Miniboxing also introduces the double factory mechanism, which pays for the reflection instantiation overhead only once, instead of doing it on each class instantiation.

9. Conclusions

We described miniboxing, a proposed extension to the specialization transformation in Scala, which reduces the bytecode generated by orders of magnitude. Miniboxing consists of the basic encoding (§3) and code transformation(§4), the runtime support (§5) and the specializing classloader (§6). Together, these techniques were able to approach the performance of monomorphic and specialized code and obtain a speedups of up to 22x over generic code (§7).

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual*. URL `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[2] L. Bourdev and J. Järvi. Efficient run-time dispatching in generic programming with minimal code bloat. *Sci. Comput. Program.*, 76(4), Apr. 2011. ISSN 0167-6423.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, 33(10), Oct. 1998. ISSN 0362-1340.

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

[5] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44. ACM, 2009.

[6] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

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

[8] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2.

[9] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1), 2008.

[10] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice*. PhD thesis, PhD thesis, Katholieke Universiteit Leuven, 2009.

[12] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(3), 1991.

[13] M. Odersky and M. Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40. ACM, 2005.

[14] M. Odersky, E. Runne, and P. Wadler. *Two ways to bake your pizza-translating parameterised types into Java*. Springer, 2000.

[15] M. Paleczny, C. Vick, and C. Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001.

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

[17] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, pages 136–147. Springer, 2011.

[18] O. Sallenave and R. Ducournau. Lightweight generics in embedded systems through static analysis. *SIGPLAN Not.*, 47(5), June 2012. ISSN 0362-1340.

[19] M. Schinz. *Compiling scala for the Java virtual machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[20] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 3rd edition, 1997. ISBN 0201889544.

[21] S. Wholey and S. E. Fahlman. The design of an instruction set for Common LISP. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3.

[22] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages*. ACM, 2012.