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

Vlad Ureche Cristian Talau Martin Odersky
EPFL, Switzerland
{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 and heterogeneous. Homogeneous translation requires boxing, and thus introduces indirect access delays. Heterogeneous translation duplicates and adapts code for each value type individually, producing more bytecode. Therefore bytecode speed and size are at odds with each other. This paper proposes a novel translation that significantly reduces the bytecode size without affecting the execution speed. The key insight is that larger value types (such as integers) can hold smaller ones (such as bytes) thus reducing the duplication necessary in heterogeneous translations. In our implementation, on the Scala compiler, we encode all primitive value types in long integers. The resulting bytecode approaches the performance of monomorphic code, matches the performance of the heterogeneous translation and obtains speedups of up to 22x over the homogeneous translation, all with modest increases in size.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism; E.2 [*Object representation*]

Keywords Miniboxing; Specialization; Data Representation; Parametric Polymorphism; Erasure; Scala; Java Virtual Machine; Bytecode; Generics

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

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. Modern 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* or *bytecode* targeting a *virtual machine*, a compiler needs to know the exact type of the parameter, so it 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, ranging from a byte to a floating point number or a pointer to a heap object, each with different 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 new code for each type used. On the other hand, *homogeneous translation*, typically done with *erasure*, 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, which leads to indirect access to values and wasteful data representation. This, in turn, slows down the program exe-

cution and increases heap requirements. 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* [44] representation. This representation can encode different types in the same unit of memory by reserving several bits to record the type and using the rest to store the value. Aside from reducing value ranges, this representation also introduces delays when *dispatching* operations, as the value and type need to be unpacked. An alternative is the *tagged union* representation [17], which does not restrict the value range but requires more heap space.

C++ [38] and the .NET Common Language Runtime [5, 20] have shown that on-demand heterogeneous translations can obtain good performance without generating significant amounts of low level code. However, this comes at a high price: C++ has taken the approach of on-demand compile-time template expansion, where compiling the use of a generic class involves instantiating the template, type checking it and generating the resulting code. This provides the best performance possible, as the instantiated template code is monomorphic, but undermines separate compilation in two ways: first, libraries need to carry source code, namely the templates themselves, to allow separate compilation, and second, multiple instantiations of the same class for the same type arguments can be created during different compilation runs, and need to be eliminated in a later linking phase. The .NET Common Language Runtime takes a load-time, on-demand approach: it compiles generics down to bytecode with type information embedded, which the virtual machine specializes, at load-time, for the type arguments. This provides good performance at the expense of more a complex virtual machine and lock-step advancements of the type system and the virtual machine implementation.

In trying to keep separate compilation and virtual machine backward compatibility, the Java programming language [23] and other statically typed JVM languages [1–4] use homogeneous translations, which sacrifice performance. Recognizing the need for execution speed, Scala *specialization* [13] allows an *annotation-driven*, *compatible* and *opportunistic* heterogeneous transformation to Java bytecode. Programmers can explicitly annotate generic code to be transformed using a heterogeneous translation, while the rest of the code is translated using boxing [10]. Specialization is a compatible transformation, in that 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* of the code during the class compilation instead of delaying their instantiation to the time they are used, like C++ does. Although in some libraries this behavior is desirable [9], generating all heterogeneous variants up front means specializing must be done cautiously so the size of the generated bytecode does not explode. To give a sense of the amount of bytecode produced by specialization, for the Scala programming language, which has 9 primitive 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, called *miniboxing*, which relies on a very simple insight to reduce the bytecode size by orders of magnitude: since larger value types (such as integers) can hold smaller value types (such as bytes), it is enough for a heterogeneous translation to generate variants for the larger value types. In our case, on the Java Virtual Machine, miniboxing reduces the number of code variants from 10 per type parameter to just 2: reference types and the largest value type in the language, the long integer. In the `Tuple3` example, miniboxing only generates 2^3 specialized variants, two orders of magnitude less bytecode than specialization. Miniboxed code is faster than homogeneous code, as data access is done directly instead of using boxing. Unlike fixnums and tagged unions, miniboxing does not attach the type information to values but to classes and methods and thus leverages the language's static type system to optimize storage. Furthermore, the full miniboxing transformation eliminates the overhead of dispatching operations by using load-time class cloning and specialization (§6). In this context, our paper makes the following contributions:

- Presents an encoding that reduces the number of variants per type parameter in heterogeneous translations (§3) and the code transformations necessary to use this encoding (§4);
- Optimizes bulk storage (arrays) in order to reduce the heap footprint and maintain compatibility to homogeneous code, produced using erasure (§5);
- Utilizes a load-time class transformation mechanism to eliminate the cost of dispatching operations on encoded values (§6).

The miniboxing encoding can reduce duplication in any heterogeneous translation, as long as the following criteria are met:

- The value types of the statically typed target language can be encoded into one or more larger value types (which we call *storage types*) - in the work presented here we use the long integer as the single storage type for all of Scala's primitive value types;

- Conversions between the value types and their storage type do not carry significant overhead (no-op conversions are preferable, but not required);
- The set of operations allowed on generic values in the language is fixed (similar to fixing the where clauses in PolyJ [8]);
- All value types have boxed representations, in order to have a common data representation between homogeneous and miniboxed code. This representation is used to ensure compatibility between the two translations.

In order to optimize the code output by the miniboxing transformation, this paper explores the interaction between value encoding and array optimization on the HotSpot Java Virtual Machine. The final miniboxing transformation, implemented as a Scala compiler plug-in[1], approaches the performance of monomorphic code, matches the performance of specialization, and obtains speedups of up to 22x over the current homogeneous translation, all with modest increases in bytecode size (§7).

The paper will first explain the specialization transformation (§2) upon which miniboxing is built. It will then go on to explain the miniboxing encoding (§3), transformation (§4), runtime support (§5) and load-time specialization (§6). It will finish by presenting the evaluation (§7), surveying the related work (§8) and concluding (§9).

2. Specialization in Scala

This section presents specialization [13], a heterogeneous translation 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;
- Undermined data locality: Seemingly contiguous memory storages, such as arrays of integers, become arrays of pointers to heap objects, which may not necessarily be 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 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 [14, 45] uses profiling as another method of triggering compatible specialization of important traces in the program.

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 class instantiations and specialized method calls (§2.3), but the injected entities are always compatible with the homogeneous translation (§2.4). However, 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 is more sophisticated [26, 34], but for the purpose of describing specialization this example is sufficient. It is also part of the benchmarks presented in the Evaluation section (§7), 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

[1] Available at `http://scala-miniboxing.org/`.

to the 8 Java primitive types, `Unit` (which is Scala's object-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 [10] 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 `Int` and `contains_J` for `Long`.

The `contains` method from the generic parent class will be inherited by all the specialized classes. But its code is generic and does not make use of primitive values, which is suboptimal. Therefore each specialized class overrides the generic `contains` and redirects it to the corresponding specialized 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. The same transformation is applied for accessors of specialized fields, such as `head` in the `ListNode` class.

2.3 Opportunistic Tree Transformation

The program code can only refer to generic classes and methods, not their specialized variants. This happens because the specialization phase, which creates the variants, runs after the type checking phase. Thus the program is checked only against the generic classes and methods. But this does not mean specialization duplicates code in vain: aside from creating the variants, specialization also injects the specialized variants in the program code.

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. Rewiring works across separate compilation, as the specialization metadata is written 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 arguments, 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 arguments are statically known:

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

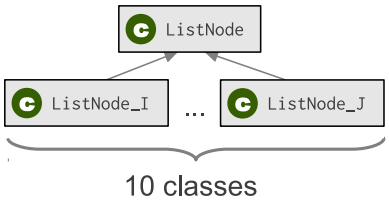

Figure 1. Class hierarchy generated by Specialization. The letters in class suffix represent the type they are specialized for: V-Scala Unit, Z-Boolean, B-Byte . . . J-Long, L-AnyRef. The names are simplified throughout the paper, and we avoid discussing the problem of name mangling, which was addressed in [13].

```
// 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 arguments. 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 arguments are 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 argument 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 specialization-annotated class `ListNode` and the type argument is statically known to be `Int`. Therefore the call to `contains` is rewired to the specialized `contains_I`:

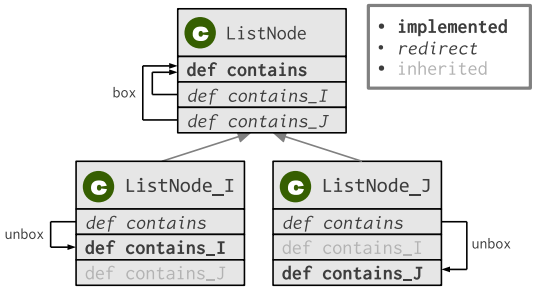

Figure 2. Method overriding and redirection for ListNode and two of its specialized variants. Constructors and accessors are omitted from this diagram.

```
// 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 known type arguments, the generic class and its specialized subclasses may be mixed together. In the following snippet, the first branch of the `if` statement is rewired to create an instance of `ListNode_I` while the second branch calls the `node` method, whose type parameter `T` is not annotated for specialization, and thus creates the generic class `ListNode`. Therefore, the value `lst` (of static type `ListNode[Int]`) may be either an instance of `ListNode_I` or of `ListNode`, depending on the random condition:

```
// 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.nextInt().isEven)
    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: The generic class at the top of the hierarchy contains all specialized variants of the `contains` method as redirects to the generic method. Then, each specialized variant of the class inherits from the generic class and overrides its corresponding specialized methods (such as `contains_I` for `ListNode_I`) with the heterogeneously transformed code and redirects the generic method to the specialized variant.

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 arguments 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.

From the perspective of typing the abstract syntax trees, compatibility is achieved because types are assigned before the specialization phase and are not modified later, so they refer to the generic class, even in the presence of rewiring. The first example in §2.3 shows that despite rewiring the **new** operator to create an instance of `ListNode_I`, the type of the `node1` value remains `ListNode[Int]`. Thus type-level compatibility is satisfied by `ListNode_I` being a subtype of `ListNode`, and the reverse subtyping is not necessary, as types never refer to `ListNode_I`[3].

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 both can be addressed by the miniboxing encoding.

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 outputs bytecode for each of them. This is done to support separate compilation.

Theoretically, the specialized variant creation could be delayed until the actual usage but this requires that the source files for specialized classes are available in all future compilation stages, exactly like in C++. This approach is undesirable from a user perspective, as it also requires encoding the original compilation flags and state, which can influence the generated code. Therefore the simplest, although bytecode-expensive solution was chosen: to generate specialized variants for all value types during compilation.

Fulfilling the bytecode compatibility requirements described before, for n type parameters and full specialization, means the generic class needs to implement 10^n methods, of which $10^n - 2$ are then inherited in the specialized subclasses and 2 are overridden by each of the 10^n subclasses. This makes the bytecode size proportional to 10^n. If the methods were not inherited but defined in each subclass, the bytecode size would be 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

[3] Except for the **this** type and singleton types in the adapted code.

specialization in `ListNode`. Experienced Scala programmers might suggest that `MyNode` should be a trait, so it can be mixed in [28]. 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 from 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 original class' constructor are not executed twice.

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.

3. Miniboxing Encoding

Constraints on the bytecode size currently prevent 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 representation 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) instead of primitive value types.

The miniboxing technique relies on a simple insight: grouping different value types reduces the number of variants necessary in the heterogeneous translation. To this end, we need to group the value types in the language into disjoint sets and for each set designate a value type, also called a storage type, which can encode any type in that set. Notice that this definition is not limited to primitive value types, but can also be used for C-like structs.

Four conditions need to be satisfied for the miniboxing transformation to work:

- All of the value types in the language can be encoded into one or more storage types;
- The overhead of transforming between any value type and its storage type must be limited, ideally a no-op;
- The operations available for generic types in the language (inherited from the top of the hierarchy, such as `toString`, `hashCode` and `equals`) must be fixed;
- All the value types need to have boxed representations, to enable compatibility between the miniboxed and homogeneous translations (§2.4). If the bytecode's common representation is tagged union, the requirement changes to having tagged union representations.

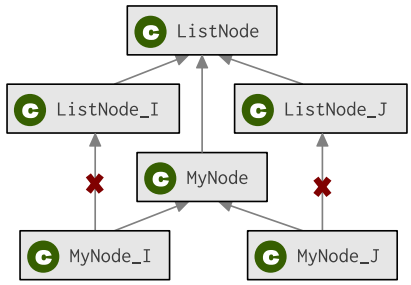

Figure 3. An example of specialized class inheritance made impossible by the current translation scheme.

In this case, the heterogeneous translation only needs to generate variants for the storage types and references. References are a special storage type, since all value types are also considered to be part of the reference group. During the translation, whenever a type is not known to be miniboxed to one of the storage types, it is automatically assumed to be attached to the references group. This allows the opportunistic (§2.3) and compatible (§2.4) rewiring of the tree: indeed since any value type has a boxed representation, it is always correct (but not optimal) to store it as a boxed reference. In the extreme case where all value types are their own storage types, we are back to specialization.

The next subsection will present miniboxing in Scala.

3.1 Miniboxing in Scala

In order to apply the miniboxing encoding to Scala, we decided to use the long integer (`Long`) as the storage type of all other primitive value types. Other sets of storage types could also be implemented to improve specific scenarios, such as running on 32-bit architectures (32-bit `Int` and 64-bit `Long`) or using floating-point numerics extensively[4] (64-bit `Double` and 64-bit `Long`). Still, for the rest of the description, we will use the long integer as the only storage type, in order to be consistent with the current implementation of the miniboxing plugin.

The transformation primitives from value types to `Long` and back are implemented in the HotSpot Java Virtual Machine and have direct translations to bytecode[4] and to processor instructions [18]. Nevertheless, two concerns need our attention when using miniboxing:

- Packing and unpacking cost;
- Memory footprint of the miniboxed encoding.

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).

[4] The floating point to integer bit-preserving transformations, which are implemented as intrinsics, do incur a measurable overhead.

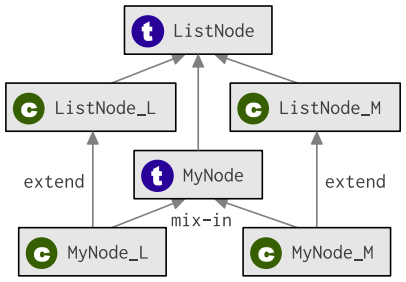

Figure 4. An example of miniboxed class inheritance. The suffixes are: M - miniboxed encoding and L - reference type. Compare to the specialized class inheritance in Figure 3.

Memory footprint. The miniboxed encoding has a memory footprint between that of monomorphic and generic code. Considering byte as the type argument, 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 (arrays) and considering the paddings introduced by the virtual machine. On the other hand, when compared to boxing 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. Therefore, all things considered, miniboxing has a memory footprint larger than the monomorphic and heterogeneous translations, but smaller than homogeneous translations based on boxing.

4. Miniboxing Transformation

The miniboxing transformation, which we developed as a Scala compiler plugin, builds upon specialization, which has been formalized in [13]. It has the same opportunistic and compatible nature and performs class and method duplication in a similar manner. Still, five elements set it apart:

- the different inheritance scheme (§4.1)
- the type bytes for storing encoded types (§4.2.1, §4.2.4)
- the use of a shallow type transformation (§4.2.2)
- the use of the final peephole transformation (§4.2.3)
- the runtime support for miniboxed values (§4.3 and §5)

4.1 Inheritance

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. As explained in §2.5, for n specialized type parameters, having a trait as the parent increases the bytecode size from 2^n to 4^n, since each of the 2^n miniboxed variants needs to implement all 2^n methods. Still, the extra bytecode is well spent, for two reasons:

- Having a trait at the top of the hierarchy means no generic fields are inherited in the specialized variants, as it happens when the homogeneous translation is at the top of the hierarchy (§2.5);
- This inheritance scheme allows specialized classes to inherit their specialized parent, thus achieving better performance in deep hierarchies.

Since the types assigned to tree nodes do not reference the specialized variants but only the generic interface, this inheritance scheme does not interfere with covariance or contravariance. Indeed, if the type parameter of `ListNode` is defined as covariant, `ListNode_M[Int]` is subtype of `ListNode[Int]` and, transitively, of `ListNode[Any]`.

4.2 Miniboxing Specifics

This section will work its way from small examples to describing the new elements in the miniboxing transformation, as compared to specialization. In order to simplify the presentation, we will use the `Long`-based encoding for miniboxing, but the transformation can still be generalized to any number of storage types.

4.2.1 Type Bytes

Type-encoding bytes (or type bytes for short) record the original type of the miniboxed values. Translating the following example shows when type bytes are necessary:

```
def print[@minispec T](value: T): Unit =
    println(value.toString)
```

Having the type parameter `T` annotated with `@minispec` will trigger miniboxing, which will duplicate this method for `Long`-encoded value types, which we also call miniboxed types. Like specialization, miniboxing produces groups of overloaded methods, with the original method being the all-reference implementation in its group. In our case, only the miniboxed overload needs to be created. To do so, the compiler will create another version of `print` for long integers, which we call `print_M`:

```
def print_M(value: Long): Unit =
    println(value.toString)
```

This is a very naive translation. Calling `print(false)`, after method rewiring, will transform the boolean to a long integer whose value will be printed on the screen instead of the "false" string. To perform the correct action, the translation should recover the string representation of the boolean value `false` from the `Long` encoding. This suggests the `toString` operation should be rewritten to:

```
def print_M(value: Long): Unit =
    println(MBRuntime.toString(value))
```

The code above shows a less naive implementation, since it rewires `toString` calls on the miniboxed value to a special runtime support object in order to obtain the string representation. But passing a single miniboxed value isn't enough, as we mentioned miniboxing does not encode the type with the value as tagged unions do [17]. Therefore, it should have a separate parameter to encode the original type:

```
def print_M(T_Type: Byte, value: Long): Unit =
    println(MBRuntime.toString(value, T_Type))
```

This is close to the minibox-transformed version of `print_M` the plugin would output. The `T_Type` field only encodes the 9 primitive types in Scala, therefore it does not incur the typical overhead of full reified generics [37]. A call to `print(false)` will be translated to the following code, where `BOOLEAN` is the type byte for boolean values:

```
print_M(BOOLEAN, MBRuntime.BoolToMinibox(false))
```

The method call above shows two differences between rewiring in miniboxing and specialization:

1. Calling a miniboxing-transformed method (or instantiating a miniboxing-transformed class) requires passing type bytes for all the `Long`-encoded type arguments;
2. The arguments to minibox-transformed methods need to be explicitly encoded in the storage type.

We will now present exactly how the miniboxing plugin arrives to this transformed code. As the miniboxing transformation takes place, it needs to preserve program semantics and type correctness. In order to do so, the transformation for `print` is actually done in three steps.

First, the new signature is created, knowing the type parameter `T` is encoded as `Long`. The method name is mangled (mangled names are simplified in this presentation) and the type byte for `T` is added to the signature. Then parameters are added, with all parameters of type `T` being replaced by parameters of type `Long`. As this happens, the symbols whose types changed are recorded and treated specially. In this case, the only miniboxed parameter is `value`, which is recorded. It is also recorded that the type byte `T_Type` corresponds to the encoded type `T`. This yields: (we'll see later why the type parameter `T` still appears)

```
def print_M[T](T_Type: Byte, value: Long): Unit
    = // need to copy and adapt body from print
```

In the second step, the body is copied from the `print` method. To maintain type correctness, all the symbols previously recorded as having their types changed are now automatically boxed back to generic type `T`, so the newly generated code tree is consistent in terms of types:

```
def print_M[T](T_Type: Byte, value: Long): Unit
    = println(MBRuntime.MiniboxToBox[T](value,
    T_Type).toString)
```

In the final step, the tree rewrite rules will transform the call to `MiniboxToBox` followed by `toString` into a single call to the `MBRuntime` system, which typically yields better performance:

```
def print_M[T](T_Type: Byte, value: Long): Unit
    = println(MBRuntime.toString(value, T_Type))
```

The next section will explain why it is necessary to carry the type parameter `T`.

4.2.2 Shallow and Deep Type Transformations

To further understand the miniboxing transformation, let us look at a more complex example, which builds a linked list with a single element:

```
def list[@minispec T](value: T): ListNode[T] =
    new ListNode[T](value, null)
```

As explained before, the `list` method will become the all-reference overload. But the interesting transformation happens in the miniboxed variant. If specialization were to transform this method its signature would be:

```
def list_M[T](value: Long): ListNode[Long]
```

The return type is incorrect, as we expect `list(3)` to return a `ListNode[Int]`, and yet rewiring `list(3)` to `list_M(...)` would return a `ListNode[Long]`. This exposes the difference between the deep type transformation in specialization and the shallow type transformation in miniboxing: In miniboxing, only values of type `T` are transformed to `Long`, but any type referring to `T`, such as `ListNode[T]`, will remain the same. This explains why the type parameter `T` is carried over to `print_M` and `list_M`: it may still be used in the method's signature and code. The full transformation for method `list_M` will be:

```
def list_M[T](T_Type: Byte, value: Long):
    ListNode[T] =
  new ListNode[T](MiniboxToBox[T](value, T_Type))
```

The shallow type transformation also changes types of local variables from `T` to `Long` and recursively transforms all nested methods and classes within the piece of code it is adapting. This propagates the storage type representation throughout the code.

4.2.3 Peephole Transformation

The last transformation to touch the code before it is shipped to the next phase is the peephole transformation, which performs a final sweep over the code to remove redundant conversions. To show this phase at work, let us consider what happens if the `ListNode` class in the last example is also annotated for miniboxing. In this case, the class will have a miniboxed variant, `ListNode_M` to which the instantiation is rewired. Since the `head` parameter of the `ListNode` constructor is boxed, while the `head` parameter of the `ListNode_M` constructor is miniboxed, the transformation will introduce a new `BoxToMinibox` conversion:

```
def list_M[T](T_Type: Byte, value: Long):
    ListNode[T] =
  new ListNode_M[T](T_Type,
      BoxToMinibox[T](MiniboxToBox[T](value,
      ...)), null)
```

Converting from `Long` to the boxed representation and back before creating the list node will certainly affect perfor-

mance. Such consecutive complementary conversions and other suboptimal constructs are automatically removed by the peephole optimization:

```
def list_M[T](T_Type: Byte, value: Long):
    ListNode[T] =
  new ListNode_M[T](T_Type, value, null)
```

The code produced by the rewiring phase can be optimized by a single pass of the peephole transformation so there is no need to iterate until a fixed point is reached.

4.2.4 Type Bytes in Classes

The class translation is slightly more complex than method translation. For classes, type bytes are also included as fields in the miniboxed variants, to allow the class' methods to encode and decode miniboxed values as necessary:

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

The interface resulting after miniboxing will be:

```
trait ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean
  def contains_M(T_Type_local: Byte, element:
      Long): Boolean
}
```

And the miniboxed variant of this class will be:

```
class ListNode_M[T]
  (T_Type: Byte, head: Long, tail: ListNode[T])
      extends ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean =
       ... // redirect to this.contains_M
  def contains_M(T_Type_local: Byte, element:
      Long): Boolean =
       ... // specialized implementation
}
```

`ListNode_M` has two type tags: `T_Type` is a class parameter and becomes a field of the class while `T_Type_local` is passed to the `contains_M` method directly. In the code example, `T_Type` is used to convert the `element` parameter of `contains` to its miniboxed representation when redirecting the call to `contains_M`. But `T_Type_local` is not used in the `ListNode_M` class. To understand when `T_Type_local` is necessary, we have to look at the reference-carrying variant of the `ListNode` class:

```
class ListNode_L[T]
  (head: T, tail: ListNode[T]) extends
      ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean =
       ... // generic implementation
  def contains_M(T_Type_local: Byte, element:
      Long): Boolean =
       ... // redirect to this.contains
}
```

All instantiations of `ListNode` where the type argument is statically known to be a value type are rewired to `ListNode_M`. The rest of the instantiations are rewired to `ListNode_L`, either because the type argument is not known statically or because it is known to be a reference type. Therefore, there is no reason for `ListNode_L` to carry `T_Type` as a global field. But, in order to allow `contains_M` to decode the miniboxed value `element` into a boxed form and redirect the call `contains`, a local type byte is necessary. Since the `ListNode` interface and its two implementations, `ListNode_L` and `ListNode_M` need to be compatible, the local type byte in `contains_M` is also present for `ListNode_M`, although in the miniboxed class it is redundant.

4.3 Calling the Runtime Support

The previous examples have shown how the miniboxing plugin uses the `MBRuntime` object for conversions between unboxed, miniboxed and boxed data representations. But the `MBRuntime` object is not limited to conversions. In Scala, any type parameter is assumed to be a subtype of the `Any` class, so the programmer can invoke methods such as `toString`, `hashCode` and `equals` on generic values. As shown in §4.2.1, these calls can be translated by a conversion to the boxed representation followed by the call, but are further optimized by calling the implementations in `MBRuntime`, which work directly on miniboxed values.

Aside from conversions and implementations for the methods in the `Any` class, the miniboxing runtime support contains code to allow direct interaction between arrays and miniboxed values. An example that uses arrays is the `ArrayBuffer` class:

```
class ArrayBuffer[@minispec T: Manifest] {
  // fields:
  private[this] var array = new Array[T](32)
  ...
  // methods:
  def getElement(idx: Int): T = array(idx)
  ...
}
```

The miniboxed variant `ArrayBuffer_M` is rewritten to call the `MBArray` object to create and access arrays in the miniboxed format:

```
// ArrayBuffer miniboxed variant for primitives:
class ArrayBuffer_M[T: Manifest](T_Type: Byte)
                        extends ArrayBuffer[T] {
  // fields:
  private[this] var array: Array[T] =
      MBArray.mbarray_new(32, T_Type)
  ...
  // methods:
  def getElement(idx: Int): T =
     MiniboxToBox(getElement_M(T_Type, idx), ...)
  def getElement_M(T_Type_local: Byte, idx:
      Int): Long =
     MBArray.array_get(array, idx, T_Type)
  ...
}
```

The implementation of the `MBArray` object is critical to numeric algorithms and performance data structures, as it has to be small enough to be inlined by the just-in-time compiler and 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 had 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 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.

In order to stay compatible with Java, Scala code uses monomorphic arrays for each value type. Therefore arrays of long integers in miniboxed classes must not be allowed to escape from the transformed class, otherwise they may crash outside code attempting to read or write them. To maintain compatibility, we could convert escaping arrays to their monomorphic forms. But the conversion would introduce delays and would break aliasing, as writes from the outside code would not be visible in the miniboxed code and vice versa. Since completely prohibiting escaping arrays severely restricts the programs that can use miniboxing, this solution becomes unusable in practice.

Thus, the only choice left is to use arrays in their monomorphic format for each value type, so we maintain compatibility with the rest of the Scala code. This decision led to 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 instruction for accessing it is used. 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 array_get[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 involved rewriting the miniboxing runtime support to match the performance of specialized code. The next subsections present the HotSpot Java Virtual Machine execution (§5.1), the main benchmark we used for testing (§5.2) and two implementations for the runtime support: type byte switching (§5.3) and object-oriented dispatching (§5.4).

5.1 HotSpot Execution

We used benchmarks to guide our implementation of the miniboxing runtime support. In this section we will briefly present the just-in-time compilation and optimization mechanisms in the HotSpot Java Virtual Machine [21, 32], since they directly influenced our design decisions. Although the work is based on the HotSpot Java Virtual Machine, we highlight the underlying mechanisms that interfere with miniboxing, in hope that our work can be used as the starting point for the analysis on different virtual machines.

The HotSpot Java Virtual Machine starts off by interpreting bytecode. After several executions, a method is considered "hot" and the just-in-time compiler is called in to transform it into native code. During compilation, aggressive inlining is done recursively on all the methods that have been both executed enough times and are small enough. Typical inlining requirements for the C2[5] (server) just-in-time compiler are 10000 executions and size below 35 bytes.

When inlining static calls, the code is inlined directly. For virtual and interface calls, however, the code depends on the receiver. To learn which code to inline, the virtual machine will profile receiver types during the interpretation phase. Then, if a single receiver is seen at runtime, the compiler will inline the method body from that receiver. This inlining 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 jumps back to interpretation mode. The bytecode may be compiled again later if it runs enough times, with both possible method bodies inlined. But if a third runtime receiver is seen, the call site is marked as megamorphic and inlining is not performed anymore, not even for the previous two method bodies.

After inlining as much code as feasible, the virtual machine's just-in-time compiler applies optimizations, which significantly reduce the running time, especially for array operations which are very regular and for which bounds checks can be eliminated.

[5] We did not use tiered compilation.

	Single Context	Multi Context
generic	20.4	21.5
miniboxed, no inlining	34.5	34.4
miniboxed, full switch	2.4	15.1
miniboxed, semi-switch	2.4	17.2
miniboxed, decision tree	24.2	24.1
miniboxed, linear	24.3	22.9
miniboxed, dispatcher	2.1	26.4
specialized	2.0	2.4
monomorphic	2.1	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).

5.2 Benchmark

We benchmarked the performance on the two examples previously shown in the paper, `ListNode` and `ArrayBuffer`. Throughout benchmarking, one particular method stood out as the most sensitive to the runtime support implementation: the `reverse` method of the `ArrayBuffer` class. The rest of this section uses the `reverse` method to explore the performance of different implementations of the runtime support:

```
def reverse_M(T_Type_local: Byte): Unit = {
  var idx = 0
  val xdi = elemCount - 1
  while (idx < xdi) {
    val el1: Long = getElement_M(T_Type, idx)
    val el2: Long = getElement_M(T_Type, xdi)
    setElement_M(T_Type, idx, el2)
    setElement_M(T_Type, xdi, el1)
    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 code 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 is benchmarked. The HotSpot Java Virtual Machine optimizations are influenced by the historical paths executed in the program, so using other type arguments can have a drastic impact on performance, as can be seen from the table, where the times for "Single Context" and "Multi Context" are very different: this means the virtual machine gives up some of its optimizations after seeing multiple instantiations with different type arguments. "Multi Context" is the likely scenario a library class will be in, as multiple instantiations with different type arguments may be created during execution.

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 just in time compiler 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 of 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 forced 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 just-in-time compiler. With all the code inlined by the Scala backend, loop unswitching was able to hoist the type byte switch statement outside the while loop. It then duplicated the loop contents for each case in the switch, allowing array-specific optimizations to bring the running time close to monomorphic code.

But using more primitive types as type arguments diminished the benefit. We tested the `reverse` operation in two situations, to check if the optimizations still take place after we use it on array buffers with different type arguments. It is frequently the case that the HotSpot 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 floating point encoding, specifically the bit-exact conversion from floating point numbers to integers, that, once executed, prevents loop unswitching.

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. What's more, the fact that the runtime "remembered" the type arguments a class was historically instantiated with made the translation unusable in practice, since this history is not only influenced by code explicitly called before the benchmark, but transitively by all code executed since the virtual machine started.

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 optimizer 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 { ... }
}
```

Ideally, this switch should be done as early as possible, even as soon as class instantiation. 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, much like the where objects in PolyJ [8]. We call this object the dispatcher. The dispatcher for each value type encodes a common set of operations such as array get and set. For example, `IntDispatcher` encodes the operations for integers:

```
abstract class Dispatcher {
  def array_get[T](arr: Array[T], idx: Int): Long
  def array_update[T](arr: Array[T], idx: Int,
      elt: Long): Unit
  ...
}
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 forced 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, dispatcher" in tables 1 and 2)

```
// 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 HotSpot 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 `array_get` and `array_set` code is not 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 profiles for each method do not interact. Accordingly, the results for integers are not influenced by the other value types used. This insight motivated the load-time cloning and specialization, which is described in the next section.

	Single Context	Multi Context
generic	20.4	21.5
miniboxed, full switch	2.4	15.1
mb. full switch, LS	2.5	**2.4**
miniboxed, dispatcher	2.1	26.4
mb. dispatcher, LS	2.0	**2.7**
specialized	2.0	2.4
monomorphic	2.1	N/A

Table 2. The time in milliseconds necessary for reversing an array buffer of 3 million integers. Miniboxing benchmarks ran with the double factory mechanism and the load-time specialization are marked with LS.

6. Miniboxing Load-time Optimization

The miniboxing runtime support, in both incarnations, using switching and dispatching, 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, as specialization does, we do it in memory, on-demand and at load-time. The .NET Common Language Runtime [5, 20] performs on-demand specialization at load-time, but it does so using more complex transformations encoded in the virtual machine. Instead, we use Java's classloading mechanism.

We use a custom classloader to clone and specialize miniboxed classes. Similar to the approach in *Pizza* [29], the classloader takes the name of a class that embeds the type byte value. For example, `ListNode_I` corresponds to a clone of `ListNode_M` with the type byte set to `INT`. From the name, the classloader infers the miniboxed class name and loads it from the classpath. It clones its bytecode and adjusts the constant table [11]. All this is done in-memory.

Once the bytecode is cloned, the paths taken through the inlined runtime support in each class remain fixed during its lifetime, making the performance in "Single Context" and "Multi Context" comparable, as can be seen in Table 2. The explanation is that the JVM sees different classes, with separate type profiles, for each primitive type.

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

- Replaces the type tag fields by static fields (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 load-time rewiring, which is described in the next section.

6.1 Miniboxing Load-time Rewiring

When rewiring, the miniboxing transformation follows the same rules set forth by specialization (§2.3). Load-time cloning introduces a new layer of rewiring, which needs to take the cloned classes into account. The factory mechanism we employ to instantiate cloned and specialized classes (§6.2) is equivalent to the instance rewiring in specialization. The two other rewiring steps in specialization are method rewiring and parent class rewiring. Fortunately method rewiring is done during compilation and since methods are not modified, there is no need to rewire them in the classloader. Parent classes, however, must be rewired at load-time to avoid performance degradation.

Load-time parent rewiring allows classes to inherit and use miniboxed methods while keeping type profiles clean. If the parent rewiring is done only at compile-time, all classes extending `ArrayBuffer_M` share the same code for the `reverse_M` method. But since they may use different type arguments when extending `ArrayBuffer`, they are back to the "Multi Context" scenario in table 1. To obtain good performance, rewiring parent classes is done first at compile time, to the miniboxed variant of the class, and then at load-time, to the cloned and specialized class. The following snippet shows parent rewiring in the case of dispatcher objects:

```
// user code:
class IntBuff extends ArrayBuffer[Int]
// after compile-time rewiring:
class IntBuff extends
    ArrayBuffer_M[Int](IntDispatcher)
// after load-time rewiring:
class IntBuff extends
    ArrayBuffer_I[Int](IntDispatcher)
```

The load-time rewiring of parent classes requires all subclasses with miniboxed parents to go through the classloader transformation. This includes the classes extending miniboxed parents with static type arguments, such as the `IntBuff` class in the code snippet before. This incurs a first-instantiation overhead, which is an inconvenience especially for classes that are only used once, such as anonymous closures extending `FunctionX`. But not all classes make use of the miniboxing runtime for arrays, so we can devise an annotation which hints to the compiler which classes need factory instantiation. This would only incur the cloning and specialization overhead when the classes use arrays. The annotation could be automatically added by the compiler when a class uses array operations and propagated from parent classes to their children:

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

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

6.2 Efficient Instantiation

Imposing the use of a global classloader is impossible in many practical applications. To allow miniboxing to work in such cases, we chose to perform the class instantiation through a factory that loads a local specializing classloader, requests the cloning and specialization of the 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 – that instantiates it using the **new** keyword. When instantiating a miniboxed class with a new set of type arguments, its corresponding factory is specialized by the classloader and instantiated via reflection. From that point on, any new instance is created by the factory, without the reflective delay. The following code snippet shows the specialized (or 2nd level) factory:

```
// 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 compiler plug-in and the miniboxing classloader (§7.1). Next, it will present the benchmarking infrastructure (§7.2) and the benchmark targets (§7.3). Finally, it will present the results (§7.4 - §7.8) and draw conclusions (§7.9).

7.1 Implementation

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 step reuses the infrastructure from specialization, with a second adaptation step which transforms storage from generic to miniboxed representation.

The plugin performs several transformations:

- Code duplication and adaptation, where values of type `T` are replaced by long integers and are un-miniboxed back to `T` at use sites (§4.2.1);

- Rewiring methods like `toString`, `hashCode`, `equals` and array operations to use the runtime support (§4.3);
- Opportunistic rewiring: new instance creation, specialized parent classes and method invocations (§2.3);
- Peephole minibox/un-minibox reduction (§4.2.3).

The miniboxing classloader duplicates classes and performs the specialized class rewiring. It uses transformations from an experimental Scala backend to perform constant propagation and dead code elimination in order to remove switches on the type byte. It supports miniboxed classes generated by the current plug-in and in the current release only works for a single specialized type parameter. Also, the infrastructure for the double factory instantiation was written and tuned by hand, and may be integrated in the plug-in in a future release. We did not implement the `@loadtimeSpec` 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 then executed by the HotSpot 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 [15].

In order to have reliable results for our microbenchmarks, we used ScalaMeter [33], a tool specifically designed to reduce benchmarking noise. ScalaMeter is currently used in performance-testing 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 (server) [32] 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, reducing the benchmark noise. For the report, we present the average of the measurements performed.

We ran the 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 Benchmark Targets

We executed the benchmarks in two scenarios:

- "Single Context" corresponds to the benchmark target (`ArrayBuffer` or `ListNode`) 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 + LS: miniboxed, type byte switching, load-time specialization with the double factory mechanism;
- mb. dispatcher + LS: miniboxed, dispatcher, load-time specialization with the double factory mechanism;
- 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 simulates collections and algorithms which make heavy use of bulk storage and the `ListNode` class simulates collections which require random heap access. We chose the benchmark methods such that each tested a certain feature of 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 array buffer 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.

`ArrayBuffer.contains` checks for the existence of elements inside an initialized array buffer. It exercises the `equals` method rewiring and revealed to 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 benchmark showed a 22x speedup over generic code.

List construction builds a 3 million element linked list using `ListNode` instances. This benchmark verifies the speed of miniboxed class instantiation. It was heavily slowed down by the reflective instantiation, 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 hash code 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 the heterogeneous translation slower.

`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 Benchmark Results

Table 3 presents the main results of our benchmarks. The table highlights "mb. switch + LS" and "mb. dispatch + LS", which represent the miniboxing encoding using the load-time specialization invoked with the double factory mechanism.

The miniboxing encoding based on type tag switching, "mb. switch + LS", offers steady performance close to that of specialization and monomorphic code, with slowdowns ranging between 0 and 20 percent. The classloader specialization, coupled with constant propagation and dead code elimination, make the type tag switching approach the most stable across multiple executions with different type arguments, with at most 6 percent difference between "Single Context" and "Multi Context", in the case of `ArrayBuffer.append`.

The dispatcher-based encoding, "mb. dispatch + LS", also offers performance close to specialization and monomorphic code, with slightly better performance when traversing the linked list (benchmarks `hashCode` and `contains`), and a lower performance on `List` creation. This suggests that passing the dispatcher object on the stack is more expensive than passing a type tag.

It is worth noting that the dispatcher-based implementation relies on inlining performed by the just-in-time compiler. Although the load-time cloning mechanism ensures type profiles remain monomorphic, the burden of inlining falls on the just-in-time compiler. In the case of virtual machines that perform ahead-of-time compilation, such as Excelsior JET [24], the newly specialized class is compiled to native code without interpretation, thus no type profiles are available and no inlining takes place for the miniboxing runtime. In contrast to dispatching, type tag switching only requires loading-time constant propagation and dead code elimination to remove the overhead of the miniboxing runtime. This makes it a better candidate for robust performance across different virtual machines. The next section will present interpreter benchmarks.

	`ArrayBuffer`			`List`		
generic	4.6	2.2	367.0	1.4	**0.2**	16.6
mb. switch + LS	**1.6**	**0.3**	**25.0**	**0.8**	**1.3**	**4.2**
mb. dispatch + LS	2.5	0.7	88.9	1.1	1.5	7.3
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 HotSpot Java Virtual Machine interpreter. The time is measured in seconds as instead of milliseconds as in the other tables. "Single context" and "Multi context" have similar results.

7.5 Interpreter Benchmarks

Before compiling the bytecode to native machine code, the HotSpot Virtual Machine interprets it and gathers profiles that later guide compilation. Table 4 presents results for running the same set of benchmarks in the interpreter, without compilation. It is important that transformations do not visibly degrade performance in the interpreter, as this slows down application startup. The data highlights a steady behavior for the the type tag switching, while the dispatcher-based approach suffers from up to 4x slowdowns.

The data shows a consistent slowdown of the tag switching approach compared to the monomorphic code in 4 of the 6 experiments. This can most likely be attributed to

	`ArrayBuffer.append`		`ArrayBuffer.reverse`		`ArrayBuffer.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	50.1	48.0	20.4	21.5	1580.1	3628.8
mb. switch	30.9	35.5	2.5	15.1	161.5	554.3
mb. dispatch	16.5	58.2	2.1	26.5	160.7	2551.6
mb. switch + LS	**15.6**	**14.8**	**2.5**	**2.4**	**159.9**	**161.7**
mb. dispatch + LS	**15.1**	**15.9**	**2.0**	**2.7**	**161.8**	**161.3**
specialization	39.7	38.5	2.0	2.4	155.8	156.3
monomorphic	16.2	N/A	2.1	N/A	157.7	N/A
	`List` creation		`List.hashCode`		`List.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	16.7	1841	22.1	20.4	1739.5	2472.4
mb. switch	11.4	11.7	18.3	18.8	1438.2	1443.2
mb. dispatch	11.4	11.5	15.6	21.0	1369.1	1753.2
mb. switch + LS	**11.5**	**11.6**	**16.2**	**16.1**	**1434.9**	**1446.3**
mb. dispatch + LS	**12.1**	**12.7**	**16.1**	**15.3**	**1364.2**	**1325.9**
specialization	11.4	11.4	14.5	36.4	1341.0	1359.2
monomorphic	10.2	N/A	13.3	N/A	1172.0	N/A

Table 3. Benchmark running times. The benchmarking setup is presented in §7.2 and the targets are presented in §7.3. The time is measured in milliseconds.

	erasure	dispatch	switch	spec.
ArrayBuffer	4.4	19.5	24.5	57.6
ArrayBuffer factory	–	+ 9.0	+ 8.5	–
ListNode	3.1	10.9	11.5	45.0
ListNode factory	–	+ 8.7	+ 8.3	–

Table 5. Bytecode generated by different translations, in kilobytes. Factories add extra bytecode for the double factory mechanism. "spec." stands for specialization.

the mechanism for invoking object methods, which requires loading a reference to the module from a static field and then performing a method call. Even after the method call is inlined, the Scala backend (and the load-time specializer) do not remove the static field access, thus leaving the redundant but possibly side-effecting instruction in the hot loop. In the native code the field access is compiled away by the just-in-time compiler. This could be improved in the Scala backend.

7.6 Bytecode Size

Table 5 presents the bytecode generated for `ArrayBuffer` and `ListNode` by 4 transformations: erasure, miniboxing with dispatcher, miniboxing with switching and specialization. The fraction of bytecode created by miniboxing, when compared to specialization, 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 $n = 1$. The reason the fraction is $4^n/10^n$ instead of $2^n/10^n$ is explained in §4.1. The double factory mechanism adds a significant bytecode, in the order of 10 kilobytes per class.

In order to evaluate the benefits of using the miniboxing encoding for real-world software, we developed a "specialization-hijacking" mode, where specialization was turned off and all `@specialized` notations were treated as `@minispec`, thus triggering miniboxing on all methods and classes where specialization was used. For this benchmark we only used the switching-based transformation.

The first evaluation was performed on Spire [31], a Scala library providing abstractions for numeric types, ranging from boolean algebras to complex number algorithms. Spire is the one library in the Scala community which uses specialization the most, and the project owner, Erik Osheim, contributed numerous bug fixes and enhancements to the Scala compiler in the area of specialization. The results, presented in Table 6, show a bytecode reduction of 2.8x and a 1.4x, or 40%, reduction in the number of specialized classes. The two reductions are not proportional because specialized methods inflate the code size of classes, but do not increase the class count. The bytecode reduction is limited to 2.8x because specialization is used in a directed manner, pointing exactly to the value types which should be specialized. So, instead of generating 10 classes per type parameter, it only generates the necessary value types. Nevertheless, even starting from manually directed specialization, the miniboxing transformation is able to further reduce the bytecode size.

The second evaluation, shown in Table 7, is motivated by a common complaint in the Scala community: that the collections in the standard library should be specialized. To perform an evaluation on collections, we sliced a part of the library around the `Vector` class and examined the impact of using the specialization and miniboxing transformations. On the approximately 64 Scala classes, traits and objects included in our slice, the bytecode reduction obtained by miniboxing compared to specialization is 4.7x. Compared to the generic `Vector`, the miniboxing code growth is 1.7x, opposed to almost 8x for specialization.

7.7 Load-time Specialization Overhead

In this section we will evaluate the overhead of the double factory mechanism. There are three types of overhead involved:

- Bytecode overhead, shown in the previous section;
- Time spent specializing and loading a class;
- Heap overhead for the classloader and factory.

We will further explore the last two sources of overhead.

7.7.1 Time Spent Specializing

Table 3, in the "`List` creation" column, shows the overhead of the double factory mechanism and class specialization is not statistically noticeable after the mechanism is warmed up. Nevertheless, it is important to understand how the mechanism behaves during a cold start, as this directly impacts an application's startup time. In this subsection we will examine the overhead for a cold start, coming from two different sources:

- The runtime class specialization;
- The cold start of the double factory mechanism.

The evaluation checks the two overheads separately: in the first experiment we only load the classes (using `Class.forName`) to trigger the runtime class specialization, while in the second experiment we instantiate the classes, either directly, using the `new` operator or through the double

	bytecode size (KB)	classes
Spire - specialized (current)	13476	2545
Spire - miniboxed	4820	1807
Spire - generic	3936	1530

Table 6. Bytecode generated by using specialization, miniboxing and leaving generic code in the Spire numeric abstractions library.

	bytecode size (KB)	classes
Vector - specialized	5691	1434
Vector - miniboxed	1210	435
Vector - generic (current)	715	223

Table 7. Bytecode generated by using specialization, miniboxing and leaving generic code on the Scala collection library slice around Vector.

	time in ms	classes
classpath - just load	182 ± 5	$9 \times 25 = 225$
classloader - warmed up	300 ± 4	225
classloader - cold start	461 ± 9	225

Table 9. Loading time (classpath) and time for cloning and specialization (classloader) for the 9 specialized variants of Vector and their transitive dependencies.

	time in ms	classes
classpath - new	258 ± 5	$9 \times 42 = 378$
classpath - factory	268 ± 6	378
classloader - factory - warm	488 ± 10	378
classloader - factory - cold	655 ± 9	378

Table 10. Instantiation time for the 9 specialized variants of Vector and their transitive dependencies.

factory mechanism. In order to evaluate the class specialization, we instrumented the specializing classloader to dump the resulting class files, such that we can compare the specializing classloader to simply loading the specialized variants from the classpath.

For the comparison, we use the `Vector` class described in the previous section. The `Vector` class mixes in 36 traits [28] which are translated by the Scala compiler as transitive dependencies of the class. In our experiments, loading the `Vector` class using `Class.forName` transitively loaded another 24 specialized classes for each variant. Instantiating a vector using `new` further loads another 18 classes, mainly specialized trait implementations and internal classes, leading to a total of 42 classes loaded with each specialized variant of `Vector`.

In each experiment we start the virtual machine, start counting the time, load or instantiate `Vector` for all 9 value types in Scala, output the elapsed time and exit. Once a class is loaded, its internal representation in the virtual machine remains cached until its classloader is garbage collected. In order to perform correct benchmarks, we chose to use a virtual machine to load the 9 specialized variants of `Vector` only once, and then restart the virtual machine. We repeated the process 100 times for each measurement.

The first experiment involves loading the class: this can be done either by using the specializing classloader to instantiate a template or by loading the class file dumped from a previous specialization run. We observed a significant difference between cold starting the specializing classloader and warming it up on a different set of classes. This is shown in Table 9: cold starting the specialization classloader incurs a slowdown of 153% while warming it up before leads to a 65% slowdown in class loading time.

The second experiment involves instantiating the class, either directly (using the `new` operator) or through the double factory mechanism. Table 10 presents the results. The surprising result of this experiment is that the overhead caused by the double factory mechanism is under 4%. As before, most of the time is spent specializing the template to produce the specialized class, which, depending on whether the classloader was used before, can lead to a slowdown between 84% and 144%. It is important to point out this overhead is a one-time cost, and further instantiations of the specialized variants take on the order of tens of milliseconds.

7.7.2 Heap Overhead

In this section we will attempt to bound the heap usage of the double factory mechanism. The double factory mechanism consists of a first level factory, which uses reflection to create second level factories, which, in turn, use the new operator to instantiate load-time specialized classes. This mechanism

	`ArrayBuffer.append`		`ArrayBuffer.reverse`		`ArrayBuffer.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	78.3	52.3	3.2	20.3	607.6	3146.1
mb. switch	27.6	×	7.4	×	844.4	×
mb. dispatch	27.0	34.8	3.2	10.8	844.7	962.7
mb. switch + LS	**22.2**	**14.3**	**3.8**	**2.9**	**725.4**	**725.2**
mb. dispatch + LS	**32.9**	**26.4**	**3.4**	**4.0**	**844.6**	**845.3**
specialization	21.7	13.4	3.5	2.7	488.7	489.4
monomorphic	19.8	N/A	3.1	N/A	490.4	N/A
	`List` creation		`List.hashCode`		`List.contains`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	32.6	23.3	13.4	13.6	1846.5	2168.1
mb. switch	23.7	18.0	11.7	10.9	1420.8	1421.5
mb. dispatch	20.9	18.3	12.4	11.4	1359.3	1427.5
mb. switch + LS	**23.2**	**17.1**	**12.2**	**10.5**	**1414.8**	**1459.4**
mb. dispatch + LS	**25.0**	**18.3**	**12.1**	**10.5**	**1390.6**	**1402.9**
specializare	21.7	16.9	12.4	10.6	1463.5	1459.8
monomorphic	19.6	N/A	11.7	N/A	1249.2	N/A

Table 8. Running times on the Graal Virtual Machine. "×" marks benchmarks for which the bytecode generated crashed the Graal just-in-time compiler. The time is measured in milliseconds.

was imposed in order to avoid the cost of reflection-based instantiation, which we found to be more expensive in terms of overhead. Each second level factory corresponds to a set of pre-determined type tags, thus instantiating two specialized variants will require two separate second level factories.

The first level factory mechanism keeps a cache of 10^n references pointing to second level factories, which is initially empty and fills up as the different variants are created. The second level factories are completely stateless and only offer a method for each specialized class constructor. Therefore the maximum heap consumption, for a 64 bit system running the HotSpot Virtual Machine, would be 16 bytes for each second level factory and 8 bytes for its cached reference, all times 10^n, assuming all variants are loaded. This means a total of 24×10^n bytes of storage. For a class with a single type parameter, this would mean a heap overhead in the order of hundreds of bytes. Assuming all of spire's specialized classes used arrays and required the two factory mechanism, since most take a single type parameter, it would mean a heap overhead in the order of tens of kilobytes.

However a hidden overhead is also present, consisting of the internal class representations for the second level factories inside the virtual machine. To bound this overhead, we can compare the factories to the classes themselves: for each specialized variant of the class there will be a specialized factory, with a method corresponding to each constructor of the class. The factory will therefore always have a strictly smaller internal representation than the specialized class, leading to at most a doubling of the internal class representation in the virtual machine.

7.8 Extending to Other Virtual Machines

In order to asses whether the miniboxing runtime system provides good performance on other virtual machines, we have evaluated it on Graal [30]. The Graal Virtual Machine consists of the same interpreter as the HotSpot Virtual Machine but a completely rewritten just-in-time compiler. Since the interpreter is the same, the same type profiles and hotness information is recorded, but the code is compiled using different transformations and heuristics. The results in Table 8 exhibit both a much lower variability but also a lower peak performance compared to the C2 compiler in HotSpot (in Table 3). With the single exception of `ArrayBuffer`'s `contains` benchmark, the switching runtime support with class loading behaves similarly to specialized code.

7.9 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, making the transformation work without any inlining support from the Scala compiler.

8. Related Work

The work by *Sallenave* and *Ducournau* [35] 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 arguments 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. 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 [10], without the need for runtime type information. Instead, type bytes are a lightweight and simple mechanism to dispatch operations for encoded value types.

According to *Morrison et al* [27] 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 but passed as separate values. 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 instructions, which makes bulk storage operations orders of magnitude faster. The initial runtime support implementations presented in §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 HotSpot Java Virtual Machine.

Fixnums in Lisp [44] 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 memory slot as the value. Although very efficient in terms of space, the fixnum representation 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 back with its type. This requires a non-negligible amount of work

for each operation. Out-of-band types are used in Lua [17], where they are implemented using tagged unions in C. Two differences set miniboxing apart: first, fixnums and tagged unions are used in homogeneous translations, whereas the miniboxing technique simplifies heterogeneous translations. Secondly, miniboxing leverages static type information to eliminate redundant type tags that would be stored in tagged unions. For example, miniboxing uses the static type information that all values in an array are of the same type: in such a case, keeping a tag for each element, as would be done with tagged unions, becomes redundant. Therefore, we consider miniboxing to be an encoding applicable to strongly typed languages, which reduces the bytecode size of heterogeneous translations, whereas fixnums and tagged unions are encodings best applied to dynamically typed languages and homogeneous translations.

The .NET Common Language Runtime [5, 20] 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 argument used. Two features are crucial in enabling this: the global presence of reified types and the instantiation mechanism in the virtual machine. Contrarily, the Java Virtual Machine does not store representations of the type arguments at runtime [10] and re-introducing them globally is very costly [37]. 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 [29] 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 the reflective instantiation overhead only once, instead of doing it on each class instantiation. The class generation from a template was first presented in the work of *Agesen et al* [6].

Around the same time as Pizza, there has been significant research on supporting polymorphism in Java, leading to work such as *GJ* [10], *NextGen* [12] and the polymorphism translation based on reflective features of *Viroli* [43]. NextGen [7, 12, 36] presents an approach where type parameter-specific operations are placed into snippet methods, which are grouped in wrapper classes, one for each polymorphic instantiation. Wrapper classes, in turn, extend a base class which contains the common functionality independent of the type parameters. It also implements a generated interface which gives the subtyping relation between the specialized classes, also supporting covariance and contravariance for the type parameters. Taking this approach of grouping common functionality in base classes, as specialization does, could reduce code duplication in miniboxed variants, at the cost of duplicating all snippet methods from the parent in the children classes. Since the collections hierarchy in Scala is up to 6 levels deep, the cost of duplicating the same snippet method 6 times outweighs the benefit of reducing local duplication in each class.

The dispatcher objects in miniboxing are specialized and restricted *where clauses* from *PolyJ* [8]. Since the methods that operate on primitive values are fixed and known a priori, unlike PolyJ, we can use dispatcher objects and type tags without any change to the virtual machine. Nevertheless it is worth noting that our implementation does pay the price of carrying dispatcher objects in each instance, which PolyJ avoids by implementing virtual machine support for invoking methods in where clauses.

In the context of ML, *Leroy* presented the idea of mixing boxed and unboxed representations of data and described the mechanism to introduce coercions between the two whenever execution passes from monomorphic to polymorphic code or back [22]. Miniboxing introduces similar coercions between the boxed and miniboxed representation, whenever the expected type is generic instead of miniboxed. The peephole optimization in miniboxing could be seen as a set of rules similar to the ones given by *Jones et al* in [19]. The work on passing explicit type representations in ML [16, 25, 41, 42] can also be seen as the base of specialization and also miniboxing. However, since we control rewiring and do it in a conservative fashion, we only use the type tags available, thus miniboxing does not need any mechanism for type argument lifting.

This paper has systematically avoided the problem of name mangling, which has been discussed in the context of Scala [13] and more recently of X10 [40]. Finally, miniboxing is not limited to classes and methods, but could also be used to reduce bytecode in specialized translations of random code blocks in the program [39].

9. Conclusions

We described miniboxing, an improved specialization transformation in Scala, which significantly reduces the bytecode generated. 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 speedups of up to 22x over the homogeneous translation (§7).

Acknowledgments

The authors would like to thank Miguel Alfredo Garcia Gutierrez for his remarks that sparked the idea of minibox-ing and for allowing early access to the experimental optimization phases from the new Scala 2.11 backend, which we used in the specializing classloader. Iulian Dragos provided invaluable help in understanding and reusing the specialization infrastructure already existing in the Scala compiler. We would also like to thank our anonymous reviewers for providing very useful feedback that completely reshaped two sections of the paper. We are also grateful to Michel Schinz, Roland Ducournau, Lukas Rytz, Vera Salvisberg, Sandro Stucki and Hubert Plociniczak who provided detailed reviews and helped us improve the paper. Last but not least, we are thankful to all the members of the Programming Languages Laboratory at EPFL (LAMP) and the Scala community, who provided a great environment for developing the miniboxing plugin, with passionate discussions, brainstorming sessions and a constant stream of good questions and quality feedback.

This research was partially supported through the European Research Council (ERC) grant 587327 “DOPPLER”.

References

[1] Ceylon Programming Language. URL `http://ceylon-lang.org/`.

[2] Kotlin Programming Language. URL `http://kotlin.jetbrains.org/`.

[3] Scala Programming Language. URL `http://scala-lang.org/`.

[4] X10 Programming Language. URL `http://x10-lang.org/`.

[5] ECMA International, Standard ECMA-335: Common Language Infrastructure, June 2006.

[6] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding Type Parameterization to the Java Language. In *ACM SIGPLAN Notices*, volume 32. ACM, 1997.

[7] E. Allen, J. Bannet, and R. Cartwright. A First-Class Approach to Genericity. In *ACM SIGPLAN Notices*, volume 38. ACM, 2003.

[8] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized Types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1997.

[9] 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.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. *SIGPLAN Notices*, 33(10), Oct. 1998. ISSN 0362-1340.

[11] A. Buckley. JSR 202: Java Class File Specification Update, 2006. URL `http://www.jcp.org/en/jsr/detail?id=202`.

[12] R. Cartwright and G. L. Steele Jr. Compatible Genericity with Run-time Types for the Java Programming Language. In *ACM SIGPLAN Notices*, volume 33. ACM, 1998.

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

[14] 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.

[15] 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, Languages and Applications*, OOPSLA ’07, 2007.

[16] R. Harper and G. Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1995.

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

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

[19] S. L. P. Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture*. Springer, 1991.

[20] 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, Snowbird, Utah, United States, 2001.

[21] 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.

[22] X. Leroy. Unboxed Objects and Polymorphic Typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1992.

[23] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin. Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP)*. ACM, 2002.

[25] Y. Minamide. Full Lifting of Type Parameters. In *Proceedings of Second Fuji International Workshop on Functional and Logic Programming*, 1997.

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

[27] 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.

[28] M. Odersky and M. Zenger. Scalable Component Abstractions. In *ACM SIGPLAN Notices*, volume 40. ACM, 2005.

[29] M. Odersky, E. Runne, and P. Wadler. *Two Ways to Bake Your Pizza-Translating Parameterised Types into Java*. Springer, 2000.

[30] Oracle. OpenJDK: Graal project. URL `http://openjdk.java.net/projects/graal/`.

[31] E. Osheim. Generic Numeric Programming Through Specialized Type Classes. *ScalaDays*, 2012.

[32] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001.

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

[34] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Euro-Par 2011 Parallel Processing*. Springer, 2011.

[35] O. Sallenave and R. Ducournau. Lightweight Generics in Embedded Systems Through Static Analysis. *SIGPLAN Notices*, 47(5), June 2012.

[36] J. Sasitorn and R. Cartwright. Efficient First-Class Generics on Stock Java Virtual Machines. In *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006.

[37] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

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

[39] N. Stucki and V. Ureche. Bridging islands of specialized code using macros and reified types. In *Proceedings of the 4th Workshop on Scala*, SCALA ’13, 2013.

[40] M. Takeuchi, S. Zakirov, K. Kawachiya, and T. Onodera. Fast Method Dispatch and Effective Use of Primitives for Reified Generics in Managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*. ACM, 2012.

[41] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996.

[42] A. Tolmach. Tag-Free Garbage Collection Using Explicit Type Parameters. In *ACM SIGPLAN Lisp Pointers*, volume 7. ACM, 1994.

[43] M. Viroli and A. Natali. Parametric Polymorphism in Java: An Approach to Translation Based on Reflective Features. In *ACM SIGPLAN Notices*, volume 35. ACM, 2000.

[44] 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, 1984.

[45] 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.