ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Master Thesis

Interoperation between Miniboxing and Other Generics Translations

Student:
Milos Stojanovic

Professor:
Martin Odersky
Supervisor:
Vlad Ureche

A thesis submitted in fulfilment of the requirements for the degree of Master in Computer Science

in the

LAMP
Computer Science

June 2015

Abstract

Generics allow programmers to design algorithms and data structures that operate in the same way regardless of the data used by abstracting over data types. Generics are useful as they improve the programmer's productivity by raising the level of abstraction, which in turn leads to reducing code duplication and uniform interfaces. However, as data on the low-level comes in different shapes and sizes, it is not a trivial job of compiler to bridge the gap between the uniform interface and the non-uniform low level implementation.

Different approaches are used for generics translation and all of them can be categorized either into homogeneous or heterogeneous group. The characteristic of homogeneous translations is that all different data representations are transformed into an identical representation and use the same low-level code for this purpose. In the heterogeneous translations, code is duplicated and adapted for each incompatible data type.

From a programmer's point of view, there should be no difference between a generic method or class compiled using some homogeneous or heterogeneous translation. Therefore, the programmer can combine different types of translations together on different parts of the code and the program has to be correct. But, as different generics translations are implemented in different ways, interoperation between them introduces noticeable slowdowns as values need to be converted to the foreign object's desired representation, incurring significant performance losses.

In this thesis, it will be explored why slowdowns happen when different translations interact together and proposed the ways how they can interoperate more efficiently. Proposed approaches are implemented and their effectiveness is presented by benchmarking the implementation.

Contents

Chapter 1

Introduction

Generics allow programmers to design algorithms and data structures that operate in the same way regardless of the data used by abstracting over data types. For example, the generic collection `Vector[T]` in the programming language Scala is expected to work identically in the case when type parameter `T` is an integer, a floating point number or any object. Generics are of crucial importance as they improve the programmer's productivity by raising the level of abstraction, which in turn leads to reducing code duplication and uniform interfaces.

However, providing a uniform interface is not trivial as data on the low-level comes in different shapes and sizes. It can be a 1-bit boolean, a 32-bit integer, a floating point number, a value class [1][2][3] or an object. For instance, in a `Vector[Int]`, getters and setters receive integer values of 32-bits while in a `Vector[String]` they receive references to heap objects. The fundamental tension here is to find a way to hide the differences between non-uniform data and to offer the uniform interface.

There are two fundamental ways of bridging the gap between non-uniform low-level data and the uniform interface exposed by generics. One is the homogeneous translation, where all different data representations are transformed into an identical representation and use the same low-level (compiled) code for this purpose. Another one is the heterogeneous translation, where the code is duplicated and adapted for each incompatible data type. Homogeneous data transformations are simpler and more common than heterogeneous, but are also less efficient as they need to convert data between different representations.

Erasure is one example of a homogeneous transformation of generics. It requires all data to be passed in by reference, pointing to a heap object, for both primitive types and objects. Erasure is the most simple and common compilation scheme for generics, but not that efficient. Requiring primitive types to be passed by reference forces the creation of heap objects in a process called boxing. Due to allocation time of objects, extra headers and garbage collection, erasure introduces additional latencies and slows down the program's execution.

Heterogeneous translations, such as specialization, are more efficient. Besides existing code which handles objects, specialization creates a separate versions of the code for each primitive type, which avoids the need for boxing. However, creating a separate version for each primitive type leads to a code explosion. Since there are 9 primitive types in Scala, for a method with a type parameter, there will be 10 versions (the reference version plus the primitive versions). If method has two type parameters, specialization creates 100 versions and in general, for N specialized type parameters, it creates 10^N specialized variants, corresponding to the Cartesian product covering all combinations. The exponential code explosion depending on the number of type parameters prevents the Scala library from using specialization extensively, since common classes have between one and three type parameters [4].

As we saw, homogeneous translations lose performance because they impose a common data format, while heterogeneous approaches produce too many versions of the code. This led to development of translation called miniboxing which is actually a hybrid of the two. It is still heterogeneous as it also handles objects and primitive types separately, but the degree of heterogeneity is significantly reduced. Unlike specialization, miniboxing does not create separate versions of the code for each existing primitive type. Instead, it encodes all primitive types on a 64-bit value, thus preventing the code explosion present in specialization.

From a programmer's point of view, there should be no difference between a generic method or class transformed using erasure, specialization or miniboxing, aside from an annotation which notifies the compiler to use one translation or another. Furthermore, the translation for each method or class can be chosen independently, without restricting the interoperability. A natural question rising here is how this can be achieved having in mind different implementations of various transformations. How all those different transformations can interoperate between themselves, work together and provide uniform interface to the programmer?

The simplest solution for interoperability between the different transformations would be passing all the data in by reference which implies encoding primitive types into heap objects. However, this solution is not efficient as it does not benefit from optimization provided by heterogeneous transformations. Another, efficient approach, would be to apply the same transformation for all parts of the code that interact together which have originally used different transformations. This is thoroughly explored in section 3.2.1. For those parts of the code where the encoding cannot be changed (such as standard libraries), there are optimized alternatives which can be introduced manually or automatically and this is explained in section 3.2.2. Thus, the contributions of this thesis are:

- exploring the ways how the different transformations can interoperate efficiently (chapter 3),
- implementing these approaches (chapter 4) and
- showing effectiveness of the approaches presented by benchmarking the implementation (chapter 5).

Chapter 2

Generics

Generics are useful as they allow abstracting over data types. Using generics, the programmer will be able to design abstract algorithms and data structures that behave identically regardless of the data used. The actual date type used by such algorithms or data structures is specified later as a parameter, when the algorithm is invoked or data structure instantiated. This permits writing common functions or types that differs only in the set of types on which they operate.

The concept of generics pioneered by ML in 1973 [29] and is supported by majority of programming languages [5]. In programming languages Ada, Delphi, Eiffel, Java, C#, F#, Swift and Visual Basic .NET this concept is known by the name generics, but other programming languages have different names for it. For example, in ML, Scala and Haskell it is also known as parametric polymorphism, and in C++ and D as templates.

The following example [6] written in Scala will demonstrate how generics can be used. Let us define generic class `Stack` which accepts one type parameter `T`:

```scala
class Stack[T] {
  private[this] var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

Generic class `Stack` can be instantiated and used in the following way:

```
object GenericsTest extends App {
  val stackInt = new Stack[Int]
  stackInt.push(1)
  stackInt.push(2)
  println(stackInt.top)
  stackInt.pop()
  println(stackInt.top)

  val stackString = new Stack[String]
  stackString.push("one")
  stackString.push("two")
  println(stackString.top)
  stackString.pop()
  println(stackString.top)
}
```

The output of this program is:

```
2
1
two
one
```

Without support for generics, we would have to implement different version of this class for each needed type separately, i.e. we would have to implement `StackInt` for integers, `StackDouble` for floating point numbers, `StackString` for `String` objects and so on. Generics helps us in a way that we can implement one class `Stack` and its methods which behave in the same way for any type used. When instantiated, the desired type would be specified as a type parameter for the class (in the example above types `Int` and `String` are used for instantiation). This way, by offering uniform interface, generics help in significantly reducing the code duplication. Also, this uniform interface raises the level of abstraction and allows programmers to think about problems on the higher level and thus improve their productivity.

Different programming languages use different data transformations in order to compile a generic code. However, almost all data transformations can be divided into two high level groups depending on the low-level code generated for generics: homogeneous and heterogeneous. The 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. This ensures good program performance, but the problem is that amount of code generated by compiling generics is huge. On the other side, the 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 execution and increases heap requirements.

In this work we will focus on generics translations available in the Scala programming language which are erasure, specialization and miniboxing.

2.1 Erasure

Erasure is the homogeneous data transformation and the current compilation scheme for generics in programming language Scala. It is one of the simplest possible compilation schemes for generics where both primitive types and objects are passed in by reference, pointing to a heap object. Consider the following implementation of method `identity` written in Scala, which accepts the parameter of type `T` and returns the same value:

```scala
def identity[T](t: T): T = t
```

The low-level (compiled) code for this method is:

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

We can see from the compiled code that method accepts and returns `Object` instead of type `T`. The erasure eliminates the type parameters and replaces all references to them by their upper bound (supertype) which is usually `Object`. Since this can invalidate a correct program, values passed to and returned from generics may need to be cast to the correct type. For example, the invocation of method `identity`:

```scala
identity[String]("x")
```

before erasure, returned a `String` and now returns an `Object`.

In the case of primitive types, such as integers, a value is converted into a heap object when it is passed to a generic code. This way primitive types are compatible with `Object` and this process is called boxing. On the other side, the return of generic methods needs to be coerced back to a primitive type from the `Object`. This inverse process is called unboxing. So, the compiled code for the invocation of the method `identity`:

```scala
val one = identity(1)
```

using erased generics would be:

```
val one: Int = identity(Integer.valueOf(1)).intValue
```

The erasure transformation is commonly used because of its simplicity, but it has several drawbacks. The problem is that boxing primitive types is an expensive operation. It requires heap allocation and garbage collection, both of which slow down program performance. Furthermore, when values are stored in generic classes, such as `Vector[T]`, they need to be stored in the boxed format, thus inflating the heap memory requirements and slowing down execution. In practice, generic methods can be as much as 10 times slower than their monomorphic (primitive) instantiations [4].

2.2 Specialization

Specialization [14][15][16] is a heterogeneous data transformation present in Scala compiler and an improvement over erasure. Specialization eliminates the overhead of type erasure while boxing the primitive types. By adding the `@specialized` annotation on type parameters, the Scala compiler specializes generics on-demand [14]. For instance, the implementation of method `identity` using specialization could be:

```
def identity[@specialized T](t: T): T = t
val one = identity(1)
```

Compiled code for this method in the case when specialization is used is:

```
def identity(t: Object): Object = t
def identity_I(t: Int): Int = t
def identity_C(t: Char): Char = t
def identity_J(t: Long): Long = t
// ... and another 6 versions of the method
```

So, the specialization creates another 9 specialized versions of method `identity` for each primitive type alone besides original generic implementation. The generic implementation may work with any object, but it requires boxing. Specialized versions of the method do not require boxing and thus they run at full speed. The compiler makes sure the implementations are “in-sync”, by deriving the specialized version from the user-defined method [14]. Instantiations of generic classes or invocations of generic methods are opportunistically rewritten to a specialized version whenever the static type indicates it is possible. On the example of the method `identity` the compiler can optimize the call to:

```
val one: Int = identity_I(1)
```

The compiler rewrites calls to generic methods to use the specialized variants of the method, if such variant exists for specified type argument. In the example above, instead of calling generic method `identity`, compiler will insert the call of method `identity_I` which corresponds to the type `Int` of method type argument and thus provide a "fast path" by avoiding the boxing and unboxing of the value.

By default, specialization is performed for all primitive types but the user may indicate that only a subset of those should be considered:

```
def identity[@specialized(Int, Char) T](t: T): T = t
```

In this case only following versions of the method are created:

```
def identity(t: Object): Object = t
def identity_I(t: Int): Int = t
def identity_C(t: Char): Char = t
```

If there is no specialized version for some type (in this case for example `Long`), generic implementation `identity` will be used and the value will be boxed and unboxed.

Since Scala specialization occurs at compile-time, all specialized variants of methods and classes appear in the generated byte code, increasing its size. This increase becomes a combinatorial explosion when there are multiple specialized type parameters since each possible combination generates a unique specialized implementation. As we have seen, there are 10 different versions of the method for only one type parameter. In general, for N specialized type parameters, there will be 10^N unique specialized variants which corresponds to the Cartesian product covering all combinations. The byte code explosion prevents the Scala library from using specialization extensively, since common classes have between one and three type parameters [4]. Thus, there is an explicit trade-off between code size and performance and it is up to the user to decide which parts of the code should be specialized.

2.3 Miniboxing

Homogeneous translations lose performance because they impose a common data format and require boxing and unboxing of values. On the other hand, heterogeneous approaches produce too many versions of the code. The miniboxing transformation [17][18] is a hybrid between the homogeneous and heterogeneous approaches, trying to minimize the issues both of them have. It is the third approach to compiling generics and can be used through a compiler plugin.

The miniboxing is motivated by the need to avoid boxing in the Scala library while also keeping the byte code growth within reasonable limits. The design of miniboxing is based on two key insights: (1) in Scala, any primitive type can be encoded in a long integer within 64 bits or double floating point number, thus reducing the duplication to two variants per type parameter and (2) the encoding requires provenance information, namely a type tag that represents the original type of the encoded value [7].

The miniboxing transformation is on-demand transformation and it is triggered in the same way as specialization by annotating the type parameter:

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

Given code is compiled to:

```
def identity(t: Object): Object = t
def identity_J(type_tag: Byte, t: Long): Long = t
def identity_D(type_tag: Byte, t: Double): Double = t
```

As we can see, two new versions of the method `identity` are created: `identity_J` and `identity_D`. The method `identity_J` will be used for integral primitive types such as `Int`, `Byte`, `Short`, `Char`, `Boolean`, `Long` and `Unit`. Another method `identity_D` will be used for floating-point primitive types: `Double` and `Float`. These methods also require a type tag corresponding to the type parameter T. The type tag is a type byte describing the type encoded in the long integer or double floating point number, allowing the operations such as `toString`, `hashCode` or `equals` to be executed correctly on encoded values, treating them as the original primitive (corresponding to `T`) rather than long integers or doubles. In the case that type `T` is any other object, generic implementation of the method is used. For all the primitive types, miniboxed version will be inserted instead of generic one and boxing and unboxing will be avoided.

The call of method `identity`:

```
val one: Int = identity(1)
```

using miniboxing transformation is compiled to:

```
val one: Int =  minibox2int(identity_J(INT, int2minibox(1)))
```

One can notice `minibox2int` and `int2minibox` transformations act exactly in the same way like boxing and unboxing in erasure transformation. However, these conversions do not create objects but merely extend values to multiple bits of representation. The values have to be coerced to the miniboxed representation, but on the Java Virtual Machine platform benchmarks have shown that the miniboxing conversion cost is completely eliminated when compiling the code to native x86 assembly [4]. Further benchmarking has shown that the code matches the performance of specialized code within a 10% slowdown due to coercions [18].

Code explosion is significantly reduced in comparison to specialization. Instead of 10 versions of the method in specialization, miniboxing creates only 3 different versions. Thus, fully specializing method with 2 parameters with miniboxing will create 3^2 different versions instead of 10^2, and in general for N type parameters there will be 3^N different versions. To conclude, miniboxing with negligible slowdowns caused by inserting the coercions in comparison to specialization, reduce the byte code generated for specializing the generics significantly.

Chapter 3

Interoperation between Generics Translations

All the different generics translations that can be used in the Scala programming language can be combined together and applied on the different parts of code. As specialization and miniboxing are triggered on demand by annotating the type parameter, and erasure is a default transformation, a situation can arise where some method with miniboxed type parameter invokes a method with specialized type parameter which invokes a method with erased type parameter:

```scala
def foo[@miniboxed T](t: T): T = bar[T](t)
def bar[@specialized T](t: T): T = baz[T](t)
def baz[T](t: T): T = t
```

In this scenario, method `foo` is compiled using miniboxing translation, method `bar` using specialization as a translation for generics and finally method `baz` using erasure. As shown in the previous section, all of the mentioned transformations are implemented in a different way so the question is how they interoperate and work together. This code is correct even though different translations are used and it produces the same result as the only one translation would be used. However, this interoperation between different translations produces noticeable slowdowns in the program execution.

In this chapter, we will concentrate on interoperation between miniboxing translation, erasure and specialization. We will explain why slow-downs happen when those translations interact together and propose techniques how to make the interoperation between them more efficient.

3.1 Miniboxing and Other Generics Translations

3.1.1 Miniboxing and Erasure

In addition to the generic implementation of some method with one type parameter, the miniboxing translation creates two new variants of the method. Calls to the miniboxed variants of the method are inserted instead of the generic method implementation when the type argument is some of the primitive types. This will ensure that miniboxed code will operate only on the miniboxed representation, without need for boxing and unboxing the value parameters.

Let us consider the following code:

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

which gets compiled to:

```
def foo(t: Object): Object = bar(t)
def foo_J(type_tag: Byte, t: Long): Long = bar_J(type_tag, t)
def foo_D(type_tag: Byte, t: Double): Double = bar_D(type_tag, t)

def bar(t: Object): Object = baz(t)
def bar_J(type_tag: Byte, t: Long): Long = baz_J(type_tag, t)
def bar_D(type_tag: Byte, t: Double): Double = baz_D(type_tag, t)

def baz(t: Object): Object = t
def baz_J(type_tag: Byte, t: Long): Long = t
def baz_D(type_tag: Byte, t: Double): Double = t
```

From the compiled code we can see that once execution entered the miniboxed path, by calling `foo_J` or `foo_D`, it continues to go through without boxing values. It just passes the encoded miniboxed representation of the value further.

But, if we break the chain and make the method `bar` use the erased generic translation:

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

the code gets compiled to:

```
def foo(t: Object): Object = bar(t)
def foo_J(type_tag: Byte, t: Long): Long =
   box2minibox(bar(minibox2box(t)))
def foo_D(type_tag: Byte, t: Double): Double =
   box2minibox(bar(minibox2box(t)))

def bar(t: Object): Object = baz(t)

def baz(t: Object): Object = t
def baz_J(type_tag: Byte, t: Long): Long = t
def baz_D(type_tag: Byte, t: Double): Double = t
```

On this example we can see both invocation of erased code from miniboxed code and invocation of miniboxed code from erased code. First case is when method `foo` invokes method `bar`. The method `foo` does not have a miniboxed version of the method `bar` to call, thus the generic version of method `bar` is invoked. Despite the fact that method `baz` is miniboxed, the value will end up boxed and then unboxed which will slow down the execution. In another case, when generic method `bar` invokes miniboxed method `baz`, boxing and unboxing of the value will happen as well. As the argument may not be primitive, generic implementation of the method `baz` has to be invoked. Thus, the optimization provided by miniboxing translation will not be exploited in this case as well and the value will end up boxed and unboxed every time.

To conclude, miniboxing transformation does not help in improving the performances of the program if it interacts with erased code. The values will end up boxed in both situations, when miniboxed code is invoked from erased code and vice-versa. As boxing cannot be avoided and miniboxing optimistic assumptions are invalidated by erased generics, the performance of the program will be the same as only erasure is applied.

3.1.2 Miniboxing and Specialization

Slow-downs in program execution occur when miniboxed code interacts with specialized code as well. As miniboxing is similar to specialization, one can expect that interaction between mentioned transformations goes smoothly and gives good performances. However, achieving efficient interaction between specialized and miniboxed code is not a trivial job.

In the following code, miniboxed method `foo` invokes specialized method `bar` which again invokes miniboxed method `baz`:

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

This code gets compiled to the following low-level code:

```
def foo(t: Object): Object = bar(t)
def foo_J(type_tag: Byte, t: Long): Long =
  box2minibox(bar(minibox2box(type_tag, t)))
def foo_D(type_tag: Byte, t: Double): Double =
  box2minibox(bar(minibox2box(type_tag, t)))

def bar(t: Object): Object = baz(t)
def bar_I(t: Int): Int = baz(t)
def bar_C(t: Char): Char = baz(t)
def bar_D(t: Double): Double = baz(t)
... // other 6 specialized variants of method bar

def baz(t: Object): Object = t
def baz_J(type_tag: Byte, t: Long): Long = t
def baz_D(type_tag: Byte, t: Double): Double = t
```

The generic variant of the method `foo` calls the generic variant of the method `bar` as there is a 1 to 1 correspondence between them. The same case is with generic variants of methods `bar` and `baz`. The miniboxed variants of method `foo` cannot call the specialized variants of method `bar` though. For instance, if type argument used when method `foo` is invoked is `Boolean`, the miniboxed variant `foo_J` will be used, but method `foo_J` does not know which specialized variant should be invoked. Therefore, value parameter has to go through processes of boxing and unboxing of value parameter when generic variant of specialized method is invoked. However, there are ways to avoid this slow path and it will be discussed in the following section.

One can say that calling miniboxed code from specialized code may work without boxing. It is clear which miniboxed variant should be invoked from specialized method variant as it is known which type is used as a type argument. However, this is not the case as miniboxing is a compiler plugin and specialization is not aware of its existence. Therefore, specialized variants will invoke generic variant of miniboxed code, where value will end up boxed and then unboxed each time.

Again, interacting between miniboxing and specialization introduces slow-downs in program executions as values go through boxing and unboxing. There are ways to make the interaction more efficient and this will be discussed in the following section.

3.1.3 Miniboxing and Scala Standard Library

Another problem is when miniboxed code interacts with libraries such as Scala standard library. Scala standard library uses either erased generics or the original specialization transformation. The problem when miniboxed code interacts with the Scala standard library is that code in Scala standard library is not in programmer's control and cannot be changed. Also, programmers are aware that most of the methods and classes in Scala standard library are specialized and expect that the specialized code invoked from miniboxed code keep good performances, but that is not the case.

For example, the following method:

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

uses erased generics versions of the tuple accessors and the function application even though type parameters `T` and `U` are annotated with miniboxing annotation. This leads to slow-downs and in the following section we will explore the approaches how to avoid this.

3.2 Efficient Interoperation

Interoperation of miniboxed code with foreign objects leads to a slow path, as values need to be converted to the foreign object's desired representation, incurring significant performance losses. Interacting with foreign objects is common and cannot be avoided, especially with the language's standard library, such as Scala standard library, which is compiled with an incompatible data encoding. Thus, it is important to find the ways how miniboxing can interoperate with foreign objects more efficiently. In previous section, it was explained why exactly slow paths occur when miniboxed code interacts with erased generics or specialized code. In this section, we will propose different approaches which can help in avoiding boxing and thus improving performances whenever it is possible.

On the high level, there are two approaches that we are proposing how interoperation of miniboxed code with erased generics or specialized code can be made more efficient. What influences the choice of the approach which can be used is whether the code that uses another translation is in programmer's control or not. If that code can be changed by the programmer, then first approach would be to show to the programmer performance advisories and guide him how to harmonize the code and use only one kind of generics translation. In the case when miniboxed code interacts with code that cannot be changed, such as Scala standard library, there are other proposed approaches: optimized accessors, wrapping objects or introducing new API.

3.2.1 Harmonizing Data Transformations

Interoperation of miniboxing with other generics translations leads to a boxing of value parameters and slow paths. Interoperation exists because different translations are applied on different parts of the code that interact together. But, in many cases there is no need for using different translations. Why then not eliminate the interoperation by harmonizing the parts of the code that are using different generics translations and use the same everywhere?

This would be possible only if all the code snippets are in programmer's control and if the choice of the translation applied can be changed by the programmer. The case when this is not possible is for example when miniboxed code interacts with some external library which is compiled using some other translation and there is no way to make that library to use miniboxing as a translation for generics.

If the code using different generics translation is in programmers control and the programmer wants to change the translation and harmonize the code, he would have to find all the places in the code when this happens. However, it is not trivial sometimes to find out when the code actually interacts with another parts of the code compiled using different translation. The case can be that programmer is not familiar with that code or that code is huge so it is difficult to search for it. If compiler silently fails to optimize the code, programmer would not be able to find that out. The performances of the program would be bad, but programmer would not have any useful information about what happened. Thus, one solution would be that compiler provides the programmer with a detailed report of the problem, exact position and advises how to harmonize the code and eliminate interoperation.

3.2.1.1 Performance Advisories

To eliminate interaction between different data transformations, compiler should point the programmer to all the situations when that happens and give the advise how to harmonize those parts of code. Let us consider again the simple example where miniboxed code interacts with erased code:

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

As explained in the previous section, value `t` ends up boxed in both situations: when miniboxed code invokes generic code and vice-versa. If this happens silently, programmer will not be aware of that and the program execution will be slowed-down. The solution for this is that compiler gives a warning to a programmer saying that value ended up boxed and gives the advice how this can be avoided. By eliminating usage of different translations and thus unnecessary interoperation between them which leads to boxing, this problem would be resolved. The advice to a programmer in this scenario would be to make method `bar` miniboxed as well. This way, only one transformation will be used and the value will be passed in the miniboxed representation without need for boxing.

Programmer should be warned in both cases, when method `foo` invokes generic method `bar` and when method `bar` invokes miniboxed method `baz`. In the first case, warning should be:

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

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

In the second case, warning should be:

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

     def bar[T](t: T): T = baz[T](t)
                              ^
```

Just by following the warnings, programmer will know where was the problem in the code and how to fix it. The warning will tell programmer exactly what to do and a position in the code where the change should be made. So, in this case, by adding miniboxing annotation to method `bar`, interoperation would be eliminated and the code would benefit from miniboxing transformation.

In another situation, when miniboxed code interacts with specialized code:

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

problem is the same as the value parameter gets boxed as explained in the previous section. Again, if the compiler warns a programmer and gives an advice how to eliminate this problem, the slow path can be avoided. The warning when method `foo` invokes specialized method `bar` should be:

```
test.scala:7: warning: Although the type parameter T of method bar
    is specialized, miniboxing and specialization communicate among
    themselves by boxing (thus, inefficiently) on all classes other
    than as FunctionX and TupleX. If you want to maximize
    performance, consider switching from specialization to
    miniboxing: '@miniboxed T':

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

and, when specialized method `bar` invokes miniboxed method `baz`:

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

     def bar[T](t: T): T = baz[T](t)
                              ^
```

The warnings say that `@specialized` annotation should be changed to `@miniboxed` for the type parameter in method `bar`. Same as in the case when miniboxed code interacts with erased generics, by following the warnings, only one transformation would be applied everywhere. Code would be harmonized and there would not be any interoperations between different translations as the only miniboxing translation would be used.

To generalize, there are two problems that can occur. One is when miniboxed code does not have miniboxed version to call. Another is when miniboxed version of the code exists, but generic version is invoked instead from generic or specialized code (as argument may not be a primitive). These two problems correspond exactly to the two of the main of performance advisories: forward and backward.

The example of forward advisories or warnings is first warning given for both examples shown above. The name forward comes from the fact that this advisory pushes the miniboxed representation from caller to callee each time the arguments need to be boxed before being passed.

The example of backward advisories or warnings is second warning given for both examples shown above. In this case, the miniboxing annotation is propagated from callee to caller and that is the reason why is it called backward.

Programmer does not have to be an expert or even familiar with the miniboxing transformation. Only by following the warnings and advises given by compiler, optimal execution of the code can be achieved by eliminating the interoperation between different translations.

3.2.1.2 Suppressing Warnings

Programmers can be aware that miniboxed code interacts with erased generic code or specialized code and due to compatibility requirements with other JVM programs or for any other reason, they do not want to change it. In this situation, warnings are not needed and there should be a way to suppress them.

A coarse-grained approach where all the warnings are turned off is not desirable as it can hide other potentially points in code which can be optimized. For this situation, miniboxing plugin offers the `@generic` annotation which can suppress both forward and backward warnings.

For example, for following piece of code, compiler will issue both forward and backward warnings:

```
scala> def foo[@miniboxed T](t: T): T = t
foo: [T](t: T)T

scala> foo[Any](1)
<console>:9: warning: Using the type argument "Any" for the
   miniboxed type parameter T of method foo is not specific enough,
   as it could mean either a primitive or a reference type. Although
   method foo is miniboxed, it won't benefit from specialization:
              foo[Any](1)
                 ^
res9: Any = 1

scala> def bar[T](t: T): T = t
bar: [T](t: T)T

scala> bar[Int](1)
<console>:9: warning: The method bar would benefit from miniboxing
   type parameter T, since it is instantiated by a primitive type.
              bar[Int](1)
                 ^
res10: Int = 1
```

If we annotate type parameter with `@generic`, warnings will be suppressed:

```
scala> def foo[@generic @miniboxed T](t: T): T = t
foo: [T](t: T)T

scala> foo[Any](1)
res9: Any = 1

scala> def bar[@generic T](t: T): T = t
bar: [T](t: T)T

scala> bar[Int](1)
res10: Int = 1
```

Also, when miniboxed code interacts with erased generics or specialized code from libraries, warnings are turned off by default and can be turned on by setting the compiler flag: `-P:minibox:warn-all`.

3.2.1.3 Data Representation Reflection

Performance advisories provide the user with compile-time warnings when code is sub-optimal, and as mentioned above, there is a mechanism to suppress them. However, it is possible that a user want to silence the warnings but still enforce strictness and define a separate run-time behavior for the class if its type argument is a primitive or reference type. To do this, run-time checks can be conducted to determine if the code has been optimized and define custom behavior to handle the situation.

The miniboxing plugin offers a window into what code is running and how it was transformed through `MbReflection API`. This API allows the programmer to check the internal state of the plugin by reflecting on the type parameters of miniboxed classes and methods. For example, given a method with a miniboxed type parameter `T`, reflection can determine, at runtime, whether `T` is miniboxed, its instantiation, and the type used to store the value:

```
scala> import MiniboxingReflection._
  import MiniboxingReflection._

scala> def foo[@miniboxed T]: String = s]"foo[T =
   ${'reifiedType[T]'}, miniboxed into a ${'storageType[T]'}]"
  foo: [T]()String

scala> foo[Int]
  res4: String = foo[T = Int, miniboxed into a Long]
```

The method `reifiedType[T]` indicates the type of type parameter instantiation while the `storageType[T]` returns the type used for encoding data. It is also possible to call the method from an erased context, in which case it will report the fact that the type parameter is a reference to a boxed object:

```
scala> def bar[T](): String = foo[T]()
  <console>:8: warning: The following code could benefit from
   miniboxing specialization if the type parameter T of method bar
   would be marked as "@miniboxed T" (it would be used to
   instantiate miniboxed type parameter T of method foo)
        def bar[T](): String = foo[T]()
                                 ^
  bar: [T]()String

scala> bar[Int]
  <console>:10: warning: The method bar would benefit from
   miniboxing type parameter T, since it is instantiated by a
   primitive type.
              bar[Int]
                 ^
  res0: String = foo[T = Reference, miniboxed into a Reference]
```

One of the usages of data representation reflection can be when class or method is always expected to be miniboxed and assertion is raised if it is not:

```
scala> class C[@miniboxed T] {
    |   assert(isMiniboxed[T], "Type parameter T of class C is not
   miniboxed!")
    | }
  defined class C

scala> def baz[T](): C = new C[T]
    java.lang.AssertionError: assertion failed: Type parameter T of
   class C is not miniboxed!
    at scala.Predef$.assert(Predef.scala:165)
    ... 34 elided
```

3.2.2 Optimized Alternatives

In the previous approach the goal was to harmonize different snippets of the code that use different generics translations and use only one translation everywhere. This is possible only if the code which translation we want to change is in programmer's control and programmer is able to change it. However, there are situations when programmer cannot change the translation and there should be workarounds how the interaction can be made more efficient.

As mentioned above, the scenario when programmer is not able to change the translation used is when for example some external library is used in the program. The code of the library cannot be changed and has to be used as is. In programming language Scala, the most used library is Scala standard library which is compiled using specialization and erased generic translation. Using specialized methods or classes from miniboxed code will lead to boxing and slow downs in program execution. So, in this section, we will propose three approaches how this interoperation can be made more efficient and show where each works best.

3.2.2.1 Optimized Accessors

One solution how miniboxed code can invoke specialized code efficiently is by using the optimized accessors. Instantiations of specialized classes and invocations of specialized methods would be preserved and stay as they are. Compiler will whenever it finds the call to specialized method, insert the invocation of appropriate optimized accessor instead.

Job of the optimized accesors would be to invoke the appropriate specialized variant of the methods based on the type argument and without boxing the value parameters.

Besides method's arguments that have to be passed to the accessors, accesors require type tag for each type parameter of the method. Type tags (describing the encoded primitive type) will be used to switch on and decide on which specialized variant should be invoked in the accessor. This approach needs to be implemented both for accessors, allowing the specialized values to be extracted directly into the miniboxed encoding and for constructors, allowing miniboxed code to instantiate specialized classes without boxing.

For instance, if miniboxed method `foo` invokes specialized method `bar`:

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

compiler will change all the calls to the method `bar` to its corresponding accessors:

```
bar[T](t) if T is Int, Long, Byte, Boolean, Short, Char or Unit ->
  accessor_bar_long[T](tagForTypeArgument(T), t)
bar[T](t) if T is Double or Float ->
  accessor_bar_double[T](tagForTypeArgument(T), t)
```

As specialized variants of the method `bar` would be:

```
def bar(t: Object): Object = ...
def bar_I(t: Int): Int = ...
def bar_J(t: Long): Long = ...
def bar_B(t: Boolean): Boolean = ...
// ... other 6 specialized variants
```

the optimized accesors would look like:

```
def bar_accessor_long[T](t_type_tag: Byte, t: Long): T = {
  t_type_tag match {
    case INT  => int2minibox(bar_I(minibox2int()))
    case LONG => long2minibox(bar_J(minibox2long()))
    ...
    case _    => box2minibox(bar(minibox2box(t)))
  }
}

def bar_accessor_double[T](t_type_tag: Byte, t: Double): T = {
  t_type_tag match {
    case DOUBLE => double2minibox(bar_D(minibox2double()))
    case FLOAT  => float2minibox(bar_F(minibox2float()))
    case _      => box2minibox(bar(minibox2box(t)))
  }
}
```

Accessors allow simple invocations of specialized variants instead of generic ones and help in avoiding boxing parameters. Good side of accessors is that they do not require any additional memory. Drawback is that switching on type arguments introduces additional overhead. For small number of type parameters, accessors are still reasonable fast, but the overhead becomes significant if there are more than 3 type parameters. One more problem with accessors is that when multiple bytes are involved in the switch, they will generate a combinatorial explosion and this can confuse Java Virtual Machine heuristics for inlining and thus lead to slow paths. But, there is a way to avoid confusing the Java Virtual Machine inlining heuristics, by extracting the operation into a static method, that we call separately. Therefore, accessors are suitable for classes with small number of type parameters and when they are not invoked many times. One example of such classes is `Tuple2` from Scala standard library. It has only two type parameters and when instantiated, fields are in most cases accessed few times.

3.2.2.2 Wrapping Objects

Wrapping objects is an another technique that can be applied to minimize the slow path caused by interaction between different generics translations. While optimized accessors have good performance when values are accessed a couple of times during the lifetime of the object, wrapping objects offer better performance when values are accessed many times during the object's lifetime.

What this approach proposes is that new miniboxed class wraps the specialized class. This means that every miniboxed object of a new class will have a pointer to a specialized object. In addition, new miniboxed class will have the miniboxed variants of all the existing methods of the specialized class which will invoke the corresponding specialized once. This approach avoids switching on the type byte in order to call wrapped function at run-time. The switch is done when the new wrapping object is created.

If there is a specialized class `C` with method `foo`:

```scala
class C[@specialized T] {
  def foo(t: T): T = t
}
```

the miniboxed wrapping class would look like:

```
class MiniboxedC[@miniboxed T] {
  val extractC: C
  def foo(t: T): T
}
```

and when compiler finds the instantiation of class `C`, it changes it to instantiation of `MiniboxedC` by invoking the bridge method with type tag. Also, all calls to methods of class `C` will be changed to calls of methods of class `MiniboxedC`.

```
def c_bridge_long[T](type_tag: T, c: C): MiniboxedC[T] = {
  type_tag match {
    case INT =>
      val c_cast = c.asInstanceOf[C[Int]]
      new MiniboxedC[Int] {
       def extractC: C[Int] = c_cast
       def foo(t: Int): Int = c_cast.foo(t)
      }
    case LONG =>
      val c_cast = c.asInstanceOf[C[Long]]
      new MiniboxedC[Long] {
       def extractC: C[Long] = c_cast
       def foo(t: Long): Long = c_cast.foo(t)
      }
    ...
    case _ =>
      new MiniboxedC[T] {
        def extractC: C[T] = c
        def foo(t: T): T = c.foo(t)
     }
  }
}

def c_bridge_double[T](type_tag: T, c: C): MiniboxedC[T] = {
  type_tag match {
    case DOUBLE =>
      val c_cast = c.asInstanceOf[C[Double]]
      new MiniboxedC[Double] {
       def extractC: C[Double] = c_cast
       def foo(t: Double): Double = c_cast.foo(t)
      }
    case FLOAT =>
      val c_cast = c.asInstanceOf[C[Float]]
      new MiniboxedC[Float] {
       def extractC: C[Float] = c_cast
       def foo(t: Float): Float = c_cast.foo(t)
      }
    case _ =>
      new MiniboxedC[T] {
        def extractC: C[T] = c
        def foo(t: T): T = c.foo(t)
     }
  }
}
```

Bridge methods will switch on the type tag and instantiate miniboxed class which methods will then invoke right specialized methods. The switching on type tag happens only once, when the class is instantiated. Later, whenever the methods are invoked, there will not be any switching and any additional overhead. This is the reason that this approach is suitable for classes which methods are invoked many times during the object's lifetime. However, instantiating a new wrapping object on the heap besides specialized one will introduce some overhead in addition to switching at the beginning, but this is amortized over many method invocations.

3.2.2.3 New API

Previous two approaches suggest invoking corresponding specialized method variants from the miniboxed code and inserting them automatically by the compiler. But, in some cases, not all the limitations can be eliminated this way. The example is Scala `Array` class, which expects `ClassTag` in order to have access to type information about `T`. For algorithms where high performance is expected, passing `ClassTag` can decrease the performances significantly.

To address the issues like this one and similar, completely new API can be introduced with miniboxed implementation of the class that we need to perform fast. Compiler can then give warnings whenever it finds the instantiation of the class that we want to change and suggest using new API which will perform better. This is done for Scala `Array` class where new `MbArray` API is implemented.

```
scala> import scala.reflect._
import scala.reflect._

scala> def foo[@miniboxed T: ClassTag] = new Array[T](10)
<console>:10: warning: Use MbArray instead of Array to eliminate the
    need for ClassTags and benefit from seamless interoperability
    with the miniboxing specialization. For more details about
    MbArrays, please check the following link:
    http://scala-miniboxing.org/arrays.html
         def foo[@miniboxed T: ClassTag] = new Array[T](10)
                                           ^
foo: [T](implicit evidence$1: scala.reflect.ClassTag[T])Array[T]

scala> def foo[@miniboxed T] = MbArray.empty[T](10)
foo: [T]=> MbArray[T]

scala>
```

Chapter 4

Implementation

In this chapter it will be explained how different proposed approaches for avoiding slow-downs caused by interaction of different generics translations are implemented. Firstly, we will explain how performance advisories are implemented. Then we will explain the implementation of miniboxed functions, miniboxed tuples and miniboxed type classes which are the applications of proposed optimized alternatives. Before starting with explanation of mentioned implementations, we will shortly describe the phases added by miniboxing plugin to the Scala compiler and how the translation is done.

4.1 Miniboxing Plugin - Phases

```
milos@milos:~/miniboxing-plugin\$ mb-scalac -Xshow-phases
      phase name  id  description
      ----------  --  -----------
  mb-ext-pre-tpe   2
  interop-inject   7
 mb-ext-hijacker  10
mb-compile-ti...  11
mb-compile-ti...  13
  mb-ext-prepare  16
  interop-bridge  17
  interop-coerce  18
  interop-commit  19
 mb-ext-post-tpe  20
  minibox-inject  21
  minibox-bridge  22
  minibox-coerce  23
  minibox-commit  24
mb-tweak-erasure  31
```

The miniboxing plugin uses LDL (Late Data Layout) [7] mechanism for generics translation. By using LDL, new phases like `Inject`, `Coerce` and `Commit` are added to the Scala compiler. Above is the list of all phases added by miniboxing plugin needed for correct data transformation (original Scala compiler phases are omitted).

The LDL is used twice by the miniboxing plugin. First time it is needed when functions are transformed into miniboxed functions. All phases related to it have a name starting with prefix `interop`. In these phases, all functions are replaced by miniboxed functions in order to achieve better performance. Another usage of LDL is needed for all other transformations demanded when type parameters are annotated with `@miniboxed` annotation (phases with name starting with prefix `minibox`). These two applications of LDL are core of miniboxing plugin. There are other helper phases added to the Scala compiler as well such as: `mb-ext-pre-tpe`, `mb-ext-hijacker`, etc (name starting with `mb-`) supporting other various features. On the following example, we will show how the syntax tree of method `foo` looks like after some of the phases and how the code is transformed using miniboxing plugin:

```scala
def foo[@miniboxed T](f: T => T, t: T): T = f(t)
```

At the end of `parser` phase, the syntax tree looks like:

```scala
def foo[@new miniboxed() T](f: Function1[T, T], t: T): T = f(t)
```

where method parameter `f` is desugared into a `scala.Function1[T, T]`. During the phases `interop-inject`, `interop-bridge`, `interop-coerce` and `interop-commit`, `Function1` is transformed into a `MiniboxedFunction1` which can be seen if we print the syntax tree after the `interop-commit` phase:

```scala
def foo[@miniboxed T](f: MiniboxedFunction1[T,T], t: T): T =
    f.apply(t)
```

In the second LDL application, new versions of the method `foo` are created: `foo$n$D` and `foo$n$J`. First one is invoked when `Double` is used for encoding data. The other one is invoked when `Long` is used for encoding data. There is also a generic version which is invoked when the type argument is erased generic. The syntax tree printed after the phase `minibox-commit` looks like:

```
def foo[@miniboxed T](f: MiniboxedFunction1[T,T], t: T): T =
    f.apply(t);

def foo$n$D[T](T$TypeTag: Byte, f: MiniboxedFunction1[T,T],
                                                  t: Double): Double =
  f.apply$DD(T$TypeTag, T$TypeTag, t);

def foo$n$J[T](T$TypeTag: Byte, f: MiniboxedFunction1[T,T], t: Long)
                                                              : Long =
  f.apply$JJ(T$TypeTag, T$TypeTag, t)
```

4.2 Harmonizing Data Transformations

4.2.1 Sub-optimal Code Warnings

The compiler may generate sub-optimal code warnings in three cases: when some method is invoked, when new object is instantiated or when definition of class/trait/object is found which extends another one. They are generated during the method rewiring decision, which decides which method to call (the miniboxed or the generic one). If the type argument used to instantiate the method is a primitive type or another miniboxed type, miniboxed version will be used. But, if not, sub-optimal code warning will be generated explaining the problem. In all the cases, list of pairs of type parameters and corresponding type arguments will be analyzed and for each pair in the list if it satisfies the conditions, the warning will be generated. The following piece of code implements the logic for generating the sub-optimal code warnings:

```
def fromTargsAllTargs(pos: Position,
                      instantiation: List[(Symbol, Type)],
                      currentOwner: Symbol,
                      pspec: PartialSpec = Map.empty): PartialSpec = {

    val mboxedTpars =
      specializationsFromOwnerChain(currentOwner).toMap ++ pspec
    val spec: List[(Symbol, SpecInfo)] =
      instantiation.map({ (pair: (Symbol, Type)) =>
        val res: (Symbol, SpecInfo) =
          (pair._1, pair._2.withoutAnnotations) match {

            case (p, tpe)
              if ScalaValueClasses.contains(tpe.typeSymbol) =>
                (new BackwardWarning(p, tpe, pos)
                  .warn(BackwardWarningEnum.PrimitiveType,
                    inLibrary = !common.isCompiledInCurrentBatch(p)))
                (p,
   Miniboxed(PartialSpec.valueClassRepresentation(tpe.typeSymbol)))

            case (p, TypeRef(_, tpar, _))
              if tpar.deSkolemize.isTypeParameter =>
                mboxedTpars.get(tpar.deSkolemize) match {
                  case Some(spec: SpecInfo) =>
                    if (spec != Boxed)
                      (new BackwardWarning(p, tpar.tpe, pos)
   .warn(BackwardWarningEnum.MiniboxedTypeParam, inLibrary =
   !common.isCompiledInCurrentBatch(p)))
                    (p, spec)

                  case None =>
                    if
   (metadata.miniboxedTParamFlag(tpar.deSkolemize) &&
   metadata.isClassStem(tpar.deSkolemize.owner) &&
   !p.isMbArrayMethod)
                      (new ForwardWarning(p, tpar.tpe, pos)
                        .warn(ForwardWarningEnum.StemClass,
   inLibrary = !common.isCompiledInCurrentBatch(p)))
                    else
                      (new ForwardWarning(p, tpar.tpe, pos)
                        .warn(ForwardWarningEnum.InnerClass,
   inLibrary = !common.isCompiledInCurrentBatch(p)))
                    (p, Boxed)
                }

            case (p, tpe) if tpe <:< AnyRefTpe =>
              (p, Boxed)

            case (p, tpe) =>
              (new ForwardWarning(p, tpe, pos)
                .warn(ForwardWarningEnum.NotSpecificEnoughTypeParam,
   inLibrary = !common.isCompiledInCurrentBatch(p)))
              (p, Boxed)
          }
        res
    })
    spec.toMap
  }
}
```

Whenever the compiler finds one of the instantiations mentioned above when warning may be generated, it will invoke the method `fromTargsAllTargs`. Method parameter `instantiations` is a list of type parameters and type arguments for current instantiation. For each pair of type parameters and arguments, it will be checked if warning should be generated and which type of it should be shown. This method is also used to produce the specialization information for the type arguments.

As explained in section 3.2.1.1, there are two types of warnings: forward and backward. Also, there are different sub-types of forward and backward warnings. When forward or backward warning is generated, the actual type of it is also specified. There are two types of backward warnings: one is when type argument is primitive type and another when type argument is miniboxed type while in both cases type parameter is not miniboxed. Forward warnings is generated when type parameter is miniboxed, but its type argument is erased generic.

```
abstract class MiniboxWarning(p: Symbol, pos: Position, inLibrary:
    Boolean) {

  def msg(): String
  def shouldWarn(): Boolean

  def warn(): Unit = if (shouldWarn) suboptimalCodeWarning(pos, msg,
    p.isGenericAnnotated, inLibrary)

  ...
}

class BackwardWarningForPrimitiveType(nonMboxedTypeParam: Symbol,
    mboxedType: Type, pos: Position, inLibrary: Boolean) extends
    MiniboxWarning(nonMboxedTypeParam, pos, inLibrary) {

  override def msg: String = s"The
    ${nonMboxedTypeParam.owner.tweakedFullString} would benefit from
    miniboxing type " + s"parameter ${nonMboxedTypeParam.nameString},
    since it is instantiated by a primitive type."

  override def shouldWarn(): Boolean = {
    !isUselessWarning(nonMboxedTypeParam.owner) &&
    !isOwnerArray(nonMboxedTypeParam, mboxedType, pos) &&
    !isSpecialized(nonMboxedTypeParam, pos, inLibrary)
  }
}
...
class ForwardWarningForInnerClass(mboxedTypeParam: Symbol,
    nonMboxedType: Type, pos: Position, inLibrary: Boolean) extends
    MiniboxWarning(mboxedTypeParam, pos, inLibrary) {

  override def msg: String = s"The following code could benefit from
    miniboxing specialization " + s"if the type parameter
    ${nonMboxedType.typeSymbol.name} of
    ${nonMboxedType.typeSymbol.owner.tweakedToString} " + s"""would
    be marked as "@miniboxed ${nonMboxedType.typeSymbol.name}" (it
    would be used to """ + s"instantiate miniboxed type parameter
    ${mboxedTypeParam.name} of
    ${mboxedTypeParam.owner.tweakedToString})"

  override def shouldWarn(): Boolean = {
    !isUselessWarning(mboxedTypeParam.owner) &&
    !isSpecialized(mboxedTypeParam, pos, inLibrary)
  }
}
...
```

Another important parameter of method `instantiations` is `currentOwner` which is used for finding `mboxedTpars`. If we have the following situation:

```
class C[T] {}
def foo[@miniboxed T](t: T) = {
  class D extends C[T]
}
```

`mboxedTpars` will help us to find up the owner chain which type argument is actually used for instantiation and issue the warning showing exact position of the type parameter:

```
scala:4: warning: The class C would benefit from miniboxing type
   parameter T, since it is instantiated by miniboxed type parameter
   T of method foo.
 class D extends C[T]
                 ^
```

4.2.2 Suppressing Warnings

The sub-optimal code warnings can be suppressed by adding the annotation `@generic` in front of the type parameter. This way, the symbol in syntax tree will have the information that the type parameter is annotated and what is the class of the annotation. So, in order to suppress the warning, when warning is generated it will be checked if the symbol corresponding to this warning has this annotation. If that is the case, the warning will not be shown to the user.

4.3 Optimized Alternatives

4.3.1 Miniboxed Functions

For implementation of miniboxed functions, the second approach which proposes wrapping the object is applied. Switching on as many as 3 type bytes with each function application incurs a significant overhead. Functions are usually created once but applied many times, so any delay in the creation amortizes over many applications. The Scala `FunctionsX` is replaced by `MbFunctionX` , where X , the function arity, is either 0, 1 or 2 (the functions with greater arity are not specialized, due to the byte code explosion problem).

The miniboxing plugin introduces three tweaked versions of the function representation for 0, 1 and 2 arguments:

```
package miniboxing.runtime

trait MiniboxedFunction0[@miniboxed +R] {
  def f: Function0[R]
  def apply(): R
}

trait MiniboxedFunction1[@miniboxed -T1, @miniboxed +R] {
  def f: Function1[T1, R]
  def apply(t1: T1): R
}

trait MiniboxedFunction2[@miniboxed -T1,
                         @miniboxed -T2,
                         @miniboxed +R] {
  def f: Function2[T1, T2, R]
  def apply(t1: T1, t2: T2): R
}
```

It automatically wraps standard Scala functions into `MiniboxedFunctions` and modifies the signatures of methods to use them:

```
cat func.scala
object Test {
  val f: Function1[Int, Int] = (x: Int) => x
  f(3)
}

mb-scalac func.scala -Xprint:minibox-commit
[[syntax trees at end of            minibox-commit]] // func.scala
package <empty> {
  object Test extends Object {
    ...
    // notice the type change:  Function1 -> MiniboxedFunction1
    val f: miniboxing.runtime.MiniboxedFunction1[Int,Int] = ...

    def f(): miniboxing.runtime.MiniboxedFunction1[Int,Int] =
      Test.this.f
    f().apply_JJ(int2minibox(3))
  }
}
```

Here is how Scala would normally encode `f`, without the miniboxing plugin:

```
val f: Function1[Int,Int] = {
  @SerialVersionUID(0) final <synthetic> class $anonfun extends
       scala.runtime.AbstractFunction1[Int,Int] with Serializable {
    ...
    final def apply(x: Int): Int = x
  }
  new <$anon: Int => Int>(): Int => Int)
}
```

The block defines an anonymous class `$anon` which extends the `Function1` trait and implements the `apply` method. Now, when the miniboxing plugin is active, this is:

```
val f: miniboxing.runtime.MiniboxedFunction1[Int,Int] = {
  @SerialVersionUID(0) final <synthetic> class $anonfun extends
        scala.runtime.AbstractFunction1[Int,Int] with Serializable {
    ...
    final def apply(x: Int): Int = x
  }
  MiniboxedFunctionBridge.this.function1_opt_bridge_long_long
    [Int, Int]
    (5, 5, (new <$anon: Int => Int>(): Int => Int))
}
```

The `MiniboxedFunctionBridge.this.function1_opt_bridge_long_long` actually transforms the instance of `Function1` into a `MiniboxedFunction1`:

```
def function1_opt_bridge_long_long[T, R](T_Tag: Byte,
                                        R_Tag: Byte,
                                        _f: Function1[T, R]):
                                 MiniboxedFunction1[T, R] =
  ((T_Tag + R_Tag * 10) match {
    case 55 /* INT + INT * 10 */ =>
      val _f_cast = _f.asInstanceOf[Function1[Int, Int]]
      new MiniboxedFunction1[Int, Int] {
        def f: Function1[Int, Int] = _f_cast
        def apply(arg1: Int): Int = _f_cast.apply(arg1)
      }
    ...
    case _ =>
      function1_bridge(_f)
  }).asInstanceOf[MiniboxedFunction1[T, R]]
```

After compiling and letting both miniboxing and specialization do their magic, the case actually looks like (simplified):

```
case 55 =>
  val _f_cast = _f.asInstanceOf[Function1[Int, Int]]
  new MiniboxedFunction1_JJ[Int, Int](INT, INT) {
    def f: Function1[Int, Int] = _f_cast
    ...
    // callee for miniboxed sites -> no boxing
    def apply_JJ(T_Tag: Byte, arg1: Long): Long =
       // call to specialized code  -> no boxing
      _f_cast.apply$mcII$sp(long2int(arg1))
```

The bridge basically wraps the `Function1` in a `MiniboxedFunction1`, offering a call site where miniboxing can call and which, when called, invokes the specialized variant, thus avoiding boxing completely. This is the change necessary for the miniboxing plugin to avoid boxing when calling functions.

The key insight is that once the `MiniboxedFunction1` was created, there is no dispatching overhead – the class created knows exactly how to call the specialized function code, such that it avoids boxing. Alternative approaches, such as changing the `apply` method to a special call would actually perform the match on each invocation, significantly slowing down execution.

4.3.2 Miniboxed Type Classes

Both wrapping object and introducing new API approaches are applied when miniboxed versions of type classes are implemented. Methods of type classes are expected to be called many times during the object lifetime and that is the reason why optimized accessors approach was not applied here. Also, cost of automation is too big so new API for them is introduced. Whenever some of the type classes is used in the code, the user will be warned that by using new miniboxed API instead program will perform better. This is implemented for following type classes from `scala.math` package: `Numeric, Ordering, Integral, Fractional` and `Ordered`. How this is implemented will be explained on the example of `MiniboxedNumeric` as implementations of other mentioned type classes are similar.

Miniboxed version of `Numeric` has all the members of class `Numeric` but miniboxed. This is achieved by adding `@miniboxed` annotation to all the type parameters of methods, traits and objects inside the class. Also, the classes that `Numeric` class extends are changed to miniboxed versions, such as `MiniboxedOrdering`, `MiniboxedIntegral` and `MiniboxedFractional`.

```
object MiniboxedNumeric {
  trait ExtraImplicits {
    implicit def infixNumericOps[@miniboxed T](x: T)(implicit num:
   MiniboxedNumeric[T]): MiniboxedNumeric[T]#Ops = new num.Ops(x)
  }
  object Implicits extends ExtraImplicits { }

  trait IntIsMbIntegral extends MiniboxedIntegral[Int] {
    val extractNumeric: Numeric[Int] = Numeric.IntIsIntegral
    val extractIntegral: Integral[Int] = Numeric.IntIsIntegral
    def plus(x: Int, y: Int): Int = x + y
    def minus(x: Int, y: Int): Int = x - y
    def times(x: Int, y: Int): Int = x * y
    def quot(x: Int, y: Int): Int = x / y
    def rem(x: Int, y: Int): Int = x % y
    def negate(x: Int): Int = -x
    def fromInt(x: Int): Int = x
    def toInt(x: Int): Int = x
    def toLong(x: Int): Long = x.toLong
    def toFloat(x: Int): Float = x.toFloat
    def toDouble(x: Int): Double = x.toDouble
  }
  implicit object IntIsMbIntegral extends IntIsMbIntegral with
   MiniboxedOrdering.IntMbOrdering
  ...
```

`MiniboxedNumeric` object has a reference to a corresponding `Numeric` object equivalent (`extractNumeric`) which makes cost of invoking generic representation as low as accessing a field:

```
trait MiniboxedNumeric[@miniboxed T] extends MiniboxedOrdering[T] {
  val extractNumeric: Numeric[T]

  def plus(x: T, y: T): T
  def minus(x: T, y: T): T
  ...
```

The warning is generated during the `interop-commit` phase when syntax tree is transformed. If any of type classes is matched with current tree and if type parameter used for instantiation is primitive type or miniboxed type then corresponding warning will be shown to the user. The code generating the warning follows:

```
override def transform(tree0: Tree): Tree = {
  ...
  tree0 match {
  ...
    case _ if (TypeClasses.contains(tree0.symbol)) =>
      val targs  = tree0.tpe.dealiasWiden.typeArgs
      assert(targs.length == 1, "targs don't match for " + tree0 +
   ": " + targs)
      val targ = targs(0)
      // warn only if the type parameter is either a primitive type
   or a miniboxed type parameter
      if (ScalaValueClasses.contains(targ.typeSymbol) ||
   targ.typeSymbol.deSkolemize.hasAnnotation(MiniboxedClass))
        minibox.suboptimalCodeWarning(tree0.pos, "Upgrade from " +
   tree0.symbol.fullName + "[" + targ + "]" + " to " +
   TypeClasses(tree0.symbol).fullName + "[" + targ + "] to benefit
   from miniboxing specialization. " )
      super.transform(tree0)
    ...
```

and the following definition of method `foo` generates the warning:

```
scala> def foo[@miniboxed T: Numeric](t: T) = t
<console>:7: warning: Upgrade from scala.math.Numeric[T] to
   miniboxing.runtime.math.MiniboxedNumeric[T] to benefit from
   miniboxing specialization.
       def foo[@miniboxed T: Numeric](t: T) = t
                          ^
foo: [T](t: T)(implicit evidence$1: Numeric[T])T
```

4.3.3 Miniboxed Tuples

`MiniboxedTuple` is an example where the approach of optimized accessors is applied. The reason why this approach is suitable for `Tuple` classes is that they are usually created just to have their components accessed a few times during their life. This is not measured rigorously, but the experience show that this is the case. Optimized accessors are implemented for `Tuple1` and `Tuple2`, as those are the only two `Tuple` classes in Scala standard library that are specialized.

Specialized `Tuple` can be instantiated in the following way:

```
def foo: Int = {
  val tpl: Tuple1[Int] = new Tuple1[Int](5)
  tpl._1
}
```

The syntax tree after `specialize` phase would be:

```
def foo: Int = {
  val tpl: (Int,) = new Tuple1$mcI$sp(5);
  tpl._1$mcI$sp()
};
```

which means that specialized variants will be used. But, if miniboxed type parameter is used to instantiate the Tuple:

```
def foo[@miniboxed T]: T = {
  val tpl: Tuple1[T] = new Tuple1[T](5)
  tpl._1
}
```

the syntax tree would be:

```
def foo[@miniboxed T](t: T): T = {
  val tpl: (T,) = new (T,)(t);
  tpl._1()
};
def foo$n$D[T](T$TypeTag: Byte, t: Double): Double = {
  val tpl: (T,) = new
    (T,)(MiniboxConversionsDouble.this.minibox2box[T](t, T$TypeTag));
  MiniboxConversionsDouble.this.box2minibox_tt[T](tpl._1(), T$TypeTag)
};
def foo$n$J[T](T$TypeTag: Byte, t: Long): Long = {
  val tpl: (T,) = new
    (T,)(MiniboxConversionsLong.this.minibox2box[T](t, T$TypeTag));
  MiniboxConversionsLong.this.box2minibox_tt[T](tpl._1(), T$TypeTag)
};
```

which means that generic versions are used and that value needs to be boxed and unboxed. To avoid this, optimized accessors for Tuple classes are implemented and invoked instead of generic ones. The resulting syntax tree using this implementation is:

```
def foo[@miniboxed T](t: T): T = {
  val tpl: (T,) = new (T,)(t);
  tpl._1()
};
def foo$n$D[T](T$TypeTag: Byte, t: Double): Double = {
  val tpl: (T,) = MiniboxedTuple.this.newTuple1_double[T](T$TypeTag,
   t);
  MiniboxedTuple.this.tuple1_accessor_1_double[T](T$TypeTag, tpl)
};
def foo$n$J[T](T$TypeTag: Byte, t: Long): Long = {
  val tpl: (T,) = MiniboxedTuple.this.newTuple1_long[T](T$TypeTag, t);
  MiniboxedTuple.this.tuple1_accessor_1_long[T](T$TypeTag, tpl)
}
```

The optimized constructors and accessors switch on type tag and decide which specialized version should be instantiated and instead of invoking generic one, right specialized version will be invoked for miniboxed type parameter. The implementation of accessor invoked instead of `Tuple1._1` when the value is encoded into `Long` is:

```
// Tuple1._1 when type parameter is encoded into Long
def tuple1_accessor_1_long[T1](T1_Tag: Byte, _t: Tuple1[T1]): Long =
  (T1_Tag) match {
    case MiniboxConstants.INT =>
      MiniboxConversions.int2minibox(_t.asInstanceOf[Tuple1[Int]]._1)
    case MiniboxConstants.LONG =>

   MiniboxConversions.long2minibox(_t.asInstanceOf[Tuple1[Long]]._1)
    case _ =>
      MiniboxConversions.box2minibox_tt(_t._1, T1_Tag)
  }
```

The constructor for `Tuple1` when miniboxed type parameter is encoded into `Long` is:

```
// new Tuple1[T]; T is encoded into Long
def newTuple1_long[T1](T1_Tag: Byte, t1: Long): Tuple1[T1] =
  ((T1_Tag) match {
    case MiniboxConstants.INT =>
      new Tuple1[Int](MiniboxConversions.minibox2int(t1))
    case MiniboxConstants.LONG =>
      new Tuple1[Long](MiniboxConversions.minibox2long(t1))
    case MiniboxConstants.CHAR =>
      new Tuple1[Char](MiniboxConversions.minibox2char(t1))
    case MiniboxConstants.BOOLEAN =>
      new Tuple1[Boolean](MiniboxConversions.minibox2boolean(t1))
    case _ =>
      new Tuple1[T1](MiniboxConversionsLong.minibox2box[T1](t1,
   T1_Tag))
  }).asInstanceOf[Tuple1[T1]]
```

The reason why some cases are missing (in `tuple1_accessor_1_long` for example `CHAR`, `BOOLEAN`, etc.) is that the accessor is specialized only for certain type parameters and not all of them. The rewiring is done in `minibox-commit` phase as in that phase the type used for encoding the value will be known and also the type tag based on type argument so it can be passed to the right accessor.

```
override def transform(tree0: Tree): Tree = {
  ...
  tree0 match {
    ...
    // match Tuple1 constructor
    case Apply(Select(New(tpt), nme.CONSTRUCTOR),
   List(MiniboxToBox(t1, _, repr1))) if mbTuple_transform &&
   (tpt.tpe.typeSymbol == Tuple1Class) =>
      val targ1 = tpt.tpe.typeArgs(0).dealiasWiden  // type argument
      val tags = minibox.typeTagTrees(currentOwner) // all type tags
      val ttag1 = tags(targ1.typeSymbol.deSkolemize)  // type tag for
   type argument
      val ctor = MbTuple1Constructors(repr1) // symbol of optimized
   constructor
      val tree1 = gen.mkMethodCall(ctor, List(targ1), List(ttag1,
   transform(t1))) // new tree
      localTyper.typed(tree1)

    // match Tuple1._1 accessor
    case BoxToMinibox(tree@Apply(Select(tuple, field), _), _, repr)
   if mbTuple_transform && tupleAccessorSymbols.contains(tree.symbol)
   && tupleFieldNames.contains(field) =>
      val targs = tuple.tpe.widen.typeArgs // list of type arguments
      assert(targs.length ==
   numberOfTargsForTupleXClass(tuple.tpe.typeSymbol), "targs don't
   match for " + tree0 + ": " + targs)
      val targ = if (field == nme._1) targs(0) else targs(1)
      val tags = minibox.typeTagTrees(currentOwner) // all type tags
      val ttag = tags(targ.typeSymbol.deSkolemize) // type tag for
   type argument
      val accessor = MbTupleAccessor(tree.symbol)(repr) // symbol of
   optimized accessor
      val tree1 = gen.mkMethodCall(accessor, targs, List(ttag,
   tuple)) // new tree
      localTyper.typed(tree1)
  ...
}
```

When constructor or accessor is matched, it will be replaced with its optimized version using the code above.

Chapter 5

Benchmarks

Operations on `RRB-Vector` are used to benchmark the implementations of different approaches proposed for eliminating slow-downs caused by interaction between different generics translations. The `RRB-Vector` data structure [31] [32] is an improvement over the immutable `Vector`, allowing it to perform well for data parallel operations. Currently, the immutable `Vector` collection in the Scala library offers very good asymptotic performance over a wide range of sequential operations, but fails to scale well for data parallel operations. The problem is the overhead of merging the partial results obtained in parallel, due to the rigid Radix-Balanced Tree, the `Vector`'s underlying structure. Contrarily, `RRB-Vector` uses Relaxed Radix-Balanced (RRB) Trees, which allows merges to occur in effectively constant time while preserving the sequential operation performance. This enables the `RRB-Vector` to scale up as we would expect when executing data parallel operations. Thanks to the parallel improvement, the `RRB-Vector` data structure is slated to replace the `Vector` implementation in the Scala library in a future release.

Micro-benchmarks are conducted using the following `RRB-Vector` operations: `builder`, `map`, `fold` and `reverse`. Besides micro-benchmarks, one macro-benchmark is implemented which tests Least Square Linear Regression (LSLR) also using `RRB-Vector`. The `builder` simply creates the `RRB-Vector` using builder:

```
// builder
val rrbVectorBuilder = RRBVector.newBuilder[Int]
var i = 0
while (i < testSize) {
  rrbVectorBuilder += i
  i += 1
}
rrbVectorBuilder.result()
```

Other micro-benchmark operations use the already created vector:

```
// map
rrbVector.map({ x => x + 1 })
// fold
rrbVector.fold(0)((r, c) => r + c)
// reverse
rrbVector.reverse
```

The macro-benchmark uses two already created vectors, `vector_x` and `vector_y`, and calculates `y` for a given `x` by applying the least square linear regression method:

```
// least square linear regression
val lslr = new MbLSLRegression(rrbVector_x, rrbVector_y)
lslr.calc_y(100.0)
```

The ScalaMeter framework [33] is used as a benchmark platform and the measurements are conducted using JDK 1.7 on the machine with processor Intel Core i7-4600U CPU @ 2.10GHz x 4 and with RAM of 12GiB. Benchmarks' results are included in Table 5.1.

A slightly modified implementation of `RRB-Vector` is used; only methods that are required for the benchmark operations are retained in the implementation, and all other unused portions are removed. Additionally, to avoid using classes and objects from the Scala standard library, all necessary code is copy-pasted directly into the implementation of `RRB-Vector`. This is done in order to enable changing generics translation of the code to miniboxing. In cases where original classes from Scala standard library are used, there is no way to add `@miniboxed` annotation to their type parameters.

Firstly, all the benchmarks are compiled and run without the miniboxing plugin, using erased generics and specialization as generics translation and without any optimization added by miniboxing plugin. The results of "Erased generics" benchmark can be found in table 5.1 and will be used to compare the results of miniboxed implementations. Afterwards, that implementation is compiled with miniboxing plugin. The goal was to make the implementation use miniboxing transformation by listening to the performance advisories given by the miniboxing plugin at compile time. After initial compilation, the compiler will show 20 warnings about how the code can be made more efficient by adding `@miniboxed` annotation and upgrading from using Array and type classes to MbArray and miniboxed type classes. When all advices are applied and the code re-compiled, there will be new 12 warnings suggesting further possible changes. In 6 steps, all the warnings can be resolved and the code should run at maximum performance. The results of benchmarks using this implementation are in the second row of the table. Just by following compiler advice, the performances of some operations are improved by greater than 50%. The only operation whose performance could not be improved by miniboxing as `reverse`, since it does not depend on generics translation. The negligible slowdown in the miniboxed version of this operation in comparison to the erased generics version can be noticed and this is due to the transformation from and to miniboxing representation. Other operations will cause boxing and unboxing of value parameter, so miniboxed version speeds-up the execution twice in such cases.

	Builder	**Map**	**Fold**	**Reverse**	**LSLR**
Erased generics	43.197 s	103.00141 s	94.11593 s	31.40347 s	4864.63784 s
Miniboxed +functions +tuples +type classes	22.86196 s	60.36632 s	42.05129 s	35.23601 s	1818.09459 s
Miniboxed +functions +tuples -type classes	25.83488 s	65.76176 s	47.55423	38.58408 s	2023.08025 s
Miniboxed +functions -tuples -type classes	23.61408 s	62.9039 s	44.56721 s	35.36282 s	7627.1253 s
Miniboxed -functions -tuples -type classes	22.17166 s	141.88773 s	150.83549 s	35.73596 s	7739.11686 s

Table 5.1: RRB-Vector operations for 5M elements

The miniboxed implementation obtained by listening to compiler advice in the form of warnings exploits all the features existing in miniboxing plugin such as using miniboxed type classes, miniboxed tuples and miniboxed functions. To demonstrate how these features influence the performances of operations used in benchmarks, we will exclude them one by one and show the results. Firstly, to use Scala's type classes instead of miniboxed type classes, we will just change all the occurrences of them in the implementation as the rewiring is not automated for them. From the results, we can see that only numbers of `LSLR` are worse which is expected because this is the only operation that uses type classes (more specific method `sum` of `RRB-Vector` uses `Numeric`). Other operation are not influenced by this change. Next, we will downgrade by using Scala's `Tuple` classes instead of `MiniboxedTuple` classes. This has significant influence on `LSLR` macro-benchmark as it uses tuples extensively in calculations and slows-down the execution almost 4 times. Finally, the miniboxed functions are changed to Scala's functions and this makes operations `map` and `fold` slower by 2-3 times. This change influences only these operations as they are using functions as parameters. What is also interesting is the difference between the implementation of erased generics and miniboxed implementation without mentioned features (miniboxed functions, tuples and type classes). Implementation using erased generics performs significantly faster for some operations (`map`, `fold` and `LSLR`). This is due to the inefficient interoperation between miniboxing translation and specialization. So, in version that uses miniboxing but without miniboxed functions, tuples and type classes, all of the data has to be transformed from miniboxing representation to a specialized representation which slows-down the execution. Implementation that uses erased generic will use the optimization provided by Scala standard library and specializes the objects of mentioned classes.

Chapter 6

Related Work

The most significant related work lies in the area of run-time profilers which can offer feedback at the language level. We would like to point the work of St-Amour on optimization feedback [35] and feature-based profiling [36]. Profiling has existed for a long time at lower levels, such as at the Java Virtual Machine level, with profilers such as YourKit [37] or the Java VisualVM [38] or the x86 assembly, with processor hardware counters.

The area of opportunistic optimizations has seen an enormous growth thanks to dynamic languages such as JavaScript, Python and Ruby, which require shape analysis and optimistic assumptions on the object format to maximize execution speed. We would like to highlight the work of Mozilla on their *Monkey JavaScript VMs [23], Google's V8 JavaScript VM and the PyPy Python virtual machine [40][39]. While this is just a short list of highlights, the Truffle compiler [41][27][42] is now a general approach to writing interpreters that make optimistic assumptions, allowing maximum performance to be achieved by partially evaluating the interpreter for the program at hand, essentially obtaining a compiled program thanks to the first Futamura projection [43].

In the area of data representation, this work assumes familiarity with specialization [14] and miniboxing [18][17]. The program transformation which enables the functions to be transformed into miniboxed functions is thoroughly discussed in [7][44]. There has been previous work on miniboxing Scala collections [45] and on unifying specialization and reified types [46]. We have also seen a revived interest in specialization in the Java community, thanks to project Valhalla, which aims at providing specialization and value class support at the virtual machine level [48][47]. In the Java 8 Micro Edition functions are also represented differently [49].

Chapter 7

Conclusion

In this thesis firstly we presented why slow-downs happen when different compilation schemes interoperate together. Then we proposed several approaches to allowing different generics compilations schemes to interoperate without incurring performance regressions. First of them is by issuing actionable performance advisories that steer programmers away from performance regressions. Other approaches assume providing alternatives to the standard library constructs that use the miniboxing encoding, thus avoiding the conversion overhead. As explained, alternatives assume the implementation of optimized accessors, wrapped objects or even introducing new API. From the benchmarks conducted on the implemented approaches for Miniboxing plugin, we showed that the performance can be improved for more than 50% if the interoperation is eliminated or just some of the alternatives applied where complete elimination is not possible.

Bibliography

[1] Scala SIP-15: Value Classes. URL http://docs.scala-lang org/sips/completed/value-classes.html.

[2] J. Gosling. The Evolution of Numerical Computing in Java - preliminary discussion on value classes. URL http://web.archive.org/web/19990202050412/http://java.sun.com/people/jag/FP.html#classes.

[3] J. Rose. Value Types and Struct Tearing. URL https://web.archive.org/web/20140320141639/https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing.

[4] V. Ureche, M. Stojanovic, R. Beguet, N. Stucki, and M. Odersky. Avoiding the Slow Path in Compiler Optimizations.

[5] Generic Programming. URL https://en.wikipedia.org/wiki/Generic_programming.

[6] Generic Classes. URL http://docs.scala-lang.org/tutorials/tour/generic-classes.html.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In OOPSLA '14. ACM, 2014.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Late Data Layout: Unifying Data Representation Transformations. In OOPSLA. ACM, 1998.

[9] X. Leroy. Unboxed Objects and Polymorphic Typing. In PoPL. ACM, 1992.

[10] S. Wholey, and S. E. Fahlman. The Design of an Instruction Set for Common Lisp. In LFP, 1984

[11] X. Leroy. Unboxed objects and polymorphic typing. In POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA, 1992), ACM, pp. 177–188.

[12] S. L. P. Jones, and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. vol. 523 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1991, pp. 636–666

[13] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. ACM Trans. Program. Lang. Syst. 13, 3 (1991), 342–371.

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

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

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

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

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

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

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

[21] B. Goetz. State of the Specialization. URL `http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html`.

[22] A. Shankar, S. S. Sastry, R. Bodik, and J. Smith. Runtime Specialization With Optimistic Heap Analysis.

[23] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages.

[24] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript Performance by Deconstructing the Type System.

[25] L. Stadler, T. Wuerthinger, and H. Mossenbock. Partial Escape Analysis and Scalar Replacement for Java.

[26] M. N. Kedlaya, B. Robatmili, C. Cascaval, and B. Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines.

[27] A. Wob, C. Wirth, D. Bonetta, C. Seaton, and C. Humer, and H. Mossenbock. An Object Storage Model for the Truffle Language Implementation Framework.

[28] G. Bracha. Generics in the Java Programming Language.

[29] K. D. Lee. Programming Languages: An Active Learning Approach. Springer Science and Business Media. pp. 9–10. ISBN 978-0-387-79422-8.

[30] L. Bourdev, and J. Jarvi. Efficient run-time dispatching in generic programming with minimal code bloat. Journal: Science of Computer Programming - SCP , vol. 76, no. 4, pp. 243-257, 2011.

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

[32] N. Stucki. Turning Relaxed Radix Balanced Vector from Theory into Practice for Scala Collections. Master Thesis, EPFL, 2015.

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

[34] M. Stojanovic. Miniboxed RRB-Vector Benchmarks. URL https://github.com/milosstojanovic/mb-benchmarks.

[35] V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen. Optimization Coaching: Optimizers Learn to Communicate with Programmers. OOPSLA'12, 2012, 10.1145/2384616.2384629.

[36] V. St-Amour, L. Andersen, and M. Felleisen. Feature-Specific Profiling. CC'15, 2015, 10.1007/978-3-662-46663-6_3.

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

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

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

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

[41] T. Wurthinger, A. Woss, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST interpreters. DLS, 2012, ACM.

[42] T. Wurthinger, C. Wimmer, A. Woss, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. Onward, 2013, ACM.

[43] Futamura, and Yoshihiko. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. Higher-Order and Symbolic Computation, 1999, 10.1023/A:1010095604496.

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

[45] A. Genet, V. Ureche, and M. Odersky. Improving the Performance of Scala Collections with Miniboxing (EPFL-REPORT-200245). LAMP, EPFL, 2014. URL http://infoscience.epfl.ch/record/200245.

[46] N. Stucki, and V. Ureche. Bridging Islands of Specialized Code Using Macros and Reified Types. SCALA, ACM, 2013.

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

[48] J. Rose. Value Types and Struct Tearing. 2014. URL https://web.archive.org/web/20140320141639/https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing.

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