Compiling Generics Through User-Directed Type Specialization

Iulian Dragos
École Polytechnique Fédérale de Lausanne, Switzerland
iulian.dragos@epfl.ch

Martin Odersky
École Polytechnique Fédérale de Lausanne, Switzerland
martin.odersky@epfl.ch

ABSTRACT

Compilation of polymorphic code through type erasure gives compact code but performance on primitive types is significantly hurt. Full specialization gives good performance, but at the cost of increased code size and compilation time. Instead we propose a mixed approach, which allows the programmer to decide what code to specialize. Our approach supports separate compilation, allows mixing of specialized and generic code, and gives very good results in practice.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Polymorphism—*implementation*; D.3.2 [**Programming Languages**]: Object-Oriented Languages; D.3.4 [**Programming Languages**]: Compilers, Code generation, Optimization

General Terms

Languages, Performance

Keywords

Boxing, Specialization

1. INTRODUCTION

Parametric polymorphism has become a standard feature in statically typed, object-oriented programming languages. Generics, or templates in C++, allow programmers to write classes and methods that operate on many types, without depending on the concrete type. This leads to shorter and more expressive programs, while retaining the advantages of static typing.

For efficiency reasons, many programming languages choose different representations for primitive types (such as `Int` or `Double`) and for reference types. This in turn makes compilation of generic code more difficult, as the same code may need to work with one representation during an invocation, and another one during the next.

ICOOOLPS '09 Genova, Italy


```
def reverse[T](a: Array[T]) {
  for (i <- 0 until a.length)
    a(i) = a(a.length - i - 1)
}
```

In this example written in Scala [9], we have a method definition that takes one type parameter `T` and one value parameter `a`, an array whose elements have type `T`. Nothing is known about type `T` when this method is defined, but the array indexing operation needs to know what is the size of one element (assuming arrays are represented as a contiguous space in memory). For example, when `T` is `Int`, element size is one word, but when `T` is `Double` the size of elements is two words.

There have been two ways to deal with this issue:

- make generic code to use a single representation for values. This is the route taken by Java and Scala and leads to compact code, but primitive values have to be *boxed* and *unboxed* as they enter or leave generic code. This leads to significant performance loss.

- generate specialized versions for each type that is used by the generic code, or a subset thereof. This is the route taken by C++ and the .NET Common Language Runtime [7]. The disadvantage is the increase in code size (code explosion), but code using generic definitions runs at full speed.

In this paper we present a new way of compiling generic code that can be both fast and compact.

- We present a solution that is a combination of the two alternatives described above (Section 2). Most generic code uses a common representation, but when performance is critical the programmer may require certain classes or methods to be specialized. Our solution supports separate compilation and allows mixing specialized and generic code.

- We have developed an implementation for Scala, and report on performance improvements and code size impact in Section 3. We show that we can achieve improvements of more than 20x, for an increase in size between 16% and 161%.

2. SPECIALIZATION

As a running example we will consider a generic definition of unary functions

```
trait Function1[+R, -T] {
  def apply(x: T): R
}
```

A trait in Scala is similar to a Java interface, except that it may contain concrete methods. Here we define a function of one parameter whose only method is `apply`, and that represents functions from values of type `T` to values of type `R`. Concrete subclasses of this trait will provide an implementation for `apply`, for specific `T` and `R`.

```
def map[A](xs: Array[A], f: A => A): Unit= {
  for (i <- 0 until xs.length)
    xs(i) = f.apply(xs(i))
}
```

Method `map` applies the given function to each element of the given array, replacing the elements of the array with the return value of `f`. The type of the second parameter is a function from type `A` to `A`, and `A => A` is a more convenient syntax for `Function1[A, A]`. The two notations are equivalent. The call to `apply` is generic, hence the elements of the array have to be boxed. Suppose we use the method to square all elements of an array

```
map[Int](a, x: Int => x * x)
```

The second argument is a function literal that is desugared by the Scala compiler to an anonymous class extending the proper function type. The function body becomes the implementation of the abstract `apply` method.

```
class anonfun extends (Int => Int) {
  def apply(x: Object): Object =
    Int.box(apply(Int.unbox(x)))
  def apply(x: Int): Int = x * x
}
map[Int](a, new anonfun)
```

The example above shows the compiler output after *type erasure*, with type parameters turned to `Object`, the type of the common representation. The anonymous class needs to box and unbox the integer value before and after executing the function body. The first `apply` method is called a *bridge* method. It serves to implement the abstract method defined in trait `Function1`. In generic code such as `map`, it is always the bridge method that is called, because the call goes through the generic trait.

In the map call `A` is instantiated to `Int`, and a more efficient representation could be used, since both the array elements and the function arguments are `Int`. However, because the `map` code is generic, elements of the array need to be boxed. Similarly, the `apply` implementation needs to unbox its argument before squaring it, and box it back before returning. The two boxing operations correspond to the two type parameters being instantiated at `Int`.

2.1 User-Driven Type Specialization

We propose a solution that combines the generic and specialized approach. Generic code is compiled using a common representation, but the programmer may require certain classes or methods to be specialized. We define an annotation on type parameters that instructs the compiler to specialize code on that type parameter. A generic class definition may generate a set of specialized classes, deriving each one from the original class using a specific combination of specialized type parameters. When generic code is used in a context where more type information is available, and there exists a specialized version, the compiler rewrites method calls and class instantiations to the specialized version.

```
trait Function1[@specialized +R, @specialized -T] {
  def apply(x: Object): Object
  def applyIntInt(x: Int): Int =
    Int.unbox(apply(Int.box(x)))
}

class anonfun extends (Int => Int) {
  def apply(x: Object): Object =
    Int.box(applyIntInt(Int.unbox(x)))
  override def applyIntInt(x: Int): Int = x * x
}

def map(xs: Array[Object], f: Object => Object) {
  for (i <- 0 to xs.length - 1)
    xs(i) = f.apply(xs(i))
}
def mapInt(xs: Array[Int], f: Int => Int) {
  for (i <- 0 until xs.length)
    xs(i) = f.applyIntInt(xs(i))
}

mapInt(a, new anonfun)
```

Figure 1: Specialized version of the map example. Definitions added by the compiler are suffixed with a type tag

For specialized and unspecialized code to work together, specialized classes have to be subtypes of the generic ones. This fact allows the compiler to safely replace an instantiation of a generic class with a specialized class when the types are known, and the code around it to work with a specialized instance *even if not specialized itself*. This is a key difference from the existing approaches, and we discuss it in Section 4.

Figure 1 shows how the original example would look had it been written in Scala with the specialization extension, together with the additional definitions generated by the compiler. Code is shown after erasure, when type parameters have been replaced by `Object` (except for arrays, which are polymorphic at the VM level). The annotations on the type parameters of `Function1` tell the compiler to specialize on those type parameters. For simplicity, we assume specialization is done only at `Int`.

The original trait has a new member, a specialized version of the `apply` method. This specialized method calls the generic `apply` method taking care of boxing and unboxing as necessary. This ensures that even when the concrete instance is not specialized, code continues to work correctly through the generic code. The concrete implementation, class `anonfun`, overrides the generic method to call the specialized version, similarly taking care of boxing. This ensures that unspecialized code working on a specialized instance still works correctly. Specialized method `applyIntInt` carries the actual implementation, and works on unboxed integers.

The `map` method is transformed similarly, and the key bit is the call to apply in the specialized version. Instead of the generic `apply` method, the call is rewritten to `applyIntInt`. Such a rewrite is possible because the type of `f` is known statically to be `Int => Int`, and the type parameters of `Function1` are specializable, therefore such a method exists.

When the argument to `f` is a specialized instance of `Int => Int`, there won't be any boxing involved. Notice that all other, less fortunate, combinations still work, but rely on boxing: `mapInt` applied to an unspecialized function will call the default implementation of `applyIntInt` which in turn calls the generic `apply`, and `map` applied to either specialized or generic versions of `anonfun` will go through the generic `apply`.

2.2 Class specialization

We first look at how a class is specialized, and assume none of its methods has specialized type parameters (see Section 2.3 for a discussion of method specialization). We assume type parameters are automatically specialized on all primitive types, but the examples will show only the case of `Int`. A way to let the user choose what types to specialize to is discussed in Section 5.

Figure 2 shows a general example of class specialization. The left part shows the code that the programmer writes, the right part shows the output of the specialization phase. Class `C` has a specialized type parameter `T`, two methods and a field. After specialization, it gets a number of additional methods. The examples show the program after erasure, in order to make boxing explicit. Both the abstract and the concrete method have a specialized variant that calls the generic version, taking care of boxing and unboxing.

A specialized subclass `CInt` is also generated, and the concrete method `nInt` is re-implemented in a context where type `T` is bound to `Int`. This may allow more precise types and more rewrites when generic methods are used inside its body. The generic method is overridden as well, and routed to the specialized implementation. A new field is defined, and the accessors are rewritten to make use of the specialized representation.

Class fields have to be treated with care, as we should not duplicate state. Specialized instances operate on the specialized fields, but unspecialized code might access generic fields. The solution is to use accessors and treat them as normal methods subject to specialization[1]. The transformation described above adds the necessary specialized variants and overrides to guarantee consistent state. In other words, accessors of generic fields are overridden in specialized subclasses to select the specialized field. This guarantees that callers use the right representation regardless of its specialization.

We now describe specialization in a general setting. After specialization, a class may have additional members, different superclasses, and a set of specialized subclasses. We classify additional members in

- *specialized variants* A specialized variant of method m is added whenever m's parameters contain a specialized type parameter of class C. Each combination of concrete types generates a new variant, which calls the original method. If m is concrete, an implementation with a specialized body is added to specialized subclasses (see below). In Figure 2 `mInt` and `nInt` are specialized variants.

- *specialized overrides* A concrete method m that overrides method m', and m' has specialized variants in its defining class, will generate a method that overrides the inherited variant. In a sense, a method and its variants need to be in sync, so overriding the original implies overriding the variant. The specialized override has the same body as the original method m. The overriding method m, by contrast, is rewritten to call the specialized override. In Figure 2, `mInt` in class `D` is a special override.

[1]Scala already uses accessors for all fields in order to allow overriding.

For explaining special overrides, we turn to class `D` in the same example. Even though class `D` has no specialized type parameters, specialization may still need to transform its definition. First, its superclass is changed from the generic version of `C[Int]` to the specialized class `CInt`. Second, method `m` overrides a generic definition in `C` that has been specialized, therefore a special override is added in `D`, containing the original body. The generic version of `m` has already been routed to the specialized one in the definition of `CInt`.

Besides methods and fields, a Scala class may define type members and classes. For type members, specialization is straightforward substitution, while for classes it is the same transformation as above.

2.3 Method Specialization

Type parameters on method definitions can be specialized by expanding the method definition. After expansion, a method definition does not take any specialized type parameters, but it may have plain type parameters, and may use specialized type parameters of the enclosing class. Class specialization as defined in Section 2.2 works on expanded members.

Consider the following:

```
def m[@specialized B >: Lo <: Hi, C](x: B, y: C): B
```

This definition two type parameters, `B` and `C`. Parameter `B` has type bounds `Lo` and `Hi`, which means that all instantiations of `B` have to be a supertype of `Lo` and a subtype of `Hi`. We can derive the expanded method definitions of some method by iterating over all combinations of its specialized parameters (only `B` in this example), keeping only those that fall between the bounds. If we assume that `Int` falls between the bounds, this gives

```
def m[B >: Lo <: Hi, C](x: B, y: C): B
def mInt[C](x: Int, y: Int): Int
```

The question is what to do otherwise, and here we distinguish two cases:

- satisfiable: The bounds of a type parameter mention a specialized type parameter of the enclosing class. We cannot conclude that any concrete type satisfies its bounds until the enclosing class is instantiated.

- conflicting: The bounds of a type parameter clearly forbid a concrete type combination.

To understand why we need this distinction, we look at the way linked lists are defined in the standard library

```
class List[@specialized A] {
  def ::[@specialized B >: A](x: B): List[A] =
    new ::(x, this)
  ..
}
```

We notice that `Int` is not a valid specialization for the cons operation (`::`), because `Int` is not a supertype of `A`,

```
abstract class C[@specialized T] {
  def m(x: T): T
  def n(x: T): T = x

  val f: T
}

class D extends C[Int] {
  def m(x: Int): Int = x * x
}
```

```
abstract class C {
  def m(x: Object): Object
  def mInt(x: Int): Int =
    Int.unbox(m(Int.box(x)))

  def n(x: Object): Object = x
  def nInt(x: Int): Int =
    Int.unbox(n(Int.box(x)))

  private val f: Object
  def f(): Object = f
  def fInt(): Int = Int.unbox(f)
}
```

```
class CInt extends C {
  def m(x: Object): Object =
    Int.box(mInt(Int.unbox(x)))
  override def n(x: Object): Object =
    Int.box(nInt(Int.unbox(x)))
  override def nInt(x: Int): Int = x

  private val fInt: Int
  def f(): Object = Int.box(fInt())
  def fInt(): Int = fInt
}

class D extends CInt {
  override def mInt(x: Int): Int = x * x
}
```

Figure 2: Class specialization

for all types `A`. However, a cons operation specialized for `Int` makes sense when working on a `List[Int]`. By noticing that `A` is also specialized, and that the constraint may be fulfilled when specializing `List`, we let expansion generate a specialized variant for `Int`.

Expansion generates only satisfiable variants. Conflicting combinations give a compile-time warning, since most probably this is not intended.

What should go into the body of expansions? We distinguish again between two cases: a valid expansion is implemented by rewriting the original body with the valid type bindings. By contrast, a satisfiable expansion cannot be implemented the same way, as that would yield type-incorrect code (remember that satisfiable methods are instantiated at types that do *not* fall between the bounds). Therefore it needs to delegate to the original method.

3. EVALUATION

We implemented user-driven type specialization in the Scala compiler and used it to rewrite parts of the standard library. Scala runs on the Java Virtual Machine [8], and has the same set of primitive types. The JVM provides arrays for performance reasons, but the operations on them are very restricted. Furthermore, they are not polymorphic, instead each primitive type has a corresponding array type[2]. However, at the language level Scala presents arrays as polymorphic collections `Array[T]` that implement high-level operations like `foreach`. The cost of this is boxing and unboxing operations.

One of the motivating examples for this work has been the ability to provide efficient array-backed collections in the standard library. It was essential that such collections can be fast on primitive types, otherwise they would be avoided. The second use case is higher order functions and function literals. In order to allow users to write concise and efficient numeric algorithms on arrays and other data structures, it is required that functions do not require boxing of primitive values.

We evaluate our approach by looking at the cost in increased size of the standard library, and at the speedups obtained on several benchmarks. We acknowledge that these are preliminary measurements, and more realistic applications need to be added to the benchmarks. For all tests we used specialization on two primitive types (`Int` and `Double`).

[2]For simplicity, we consider all reference types to be Object. Casts are very fast in today's JVMs [5]

	Generic	Specialized	Increase
stdlib	10.7M	12.5M	17%
specific	142K	426K	300%
pack200-spec	32.1K	51.8K	161%

Figure 3: Code size

3.1 Code Size

We first discuss the cost in code size. We measure the size of the standard library with and without specialization enabled, and the increase in size of the modified classes alone.

Figure 3 shows the results of these measurements. The first line shows the increase in size of the standard library when specializing array-backed collections, functions up to two parameters and lists. The cost is relatively small, 17%. However, when we look at the modified classes alone the picture is rather bleak: the specialized code is around three times larger. This is the exact size on disk, and we believe the effect of specialization is seriously aggravated by the class file format of the JVM. Most specialized classes are relatively small, and pay the price of a full constant pool and structure information. In the last line we show the results after using the pack200 jar file compressor, which is known to share constant pool information between classfiles; the relative cost went down to almost half of what it was before.

3.2 Speedup

We measure the speed improvement on two mini benchmarks, designed to exercise the array implementation and higher order functions. All measurements are done on an Intel Core 2 Duo machine with 4 GB of RAM, running a Linux 2.6.24 kernel. The Java Virtual Machine is the HotSpot server VM, version 1.6.0_07.

The `funs` benchmark iterates over an array of 1000 elements and performs an operation on each of them. The operation is taken as parameter by a payload function, and the type of this parameter is `Int => Int`. We use a plain Java array, so that the results of this benchmark show the improvement we can expect for higher-order functions alone.

The second benchmark, `arrays`, uses a similar algorithm, but this time the array is a generic Scala collection backed by a Java array. The operation to be performed is inlined, so there is no improvement due to higher-order function specialization. This benchmark shows what improvements we

Test	Specialized (ms)	Generic (ms)	Speedup (Nx)
funs	71	1645	22.88
arrays	404	16,305	40.3

Figure 4: Steady-state performance

can expect from array specialization alone.

3.2.1 Steady-state Performance

We start by showing the improvements we get when the JVM is in a warm state. The benchmarks are run for 10 times in the same JVM, forcing a garbage collection before each iteration. We drop the first two measurements, which are likely to contain the startup time (class loading and JIT compilation). We average the remaining 8 iterations. Each such setup is run 5 times, and we average again on the results of each VM invocation.

Figure shows that the improvements are very large, 22 times for functions and 40 times for arrays. Standard deviation for the `funs` benchmark were around 1% of the average, except one invocation when it was much higher, 10%. The `arrays` benchmark had standard deviations between 1% and 8% of the average, per invocation. Due to the very large difference between the two alternatives, we didn't feel necessary to conduct more sophisticated statistical analysis.

3.2.2 Start-up Time

For measuring the start-up time we consider how much longer did the first iteration of each benchmark take, compared to the average over the last 8 iterations. We expected to see an increase in startup time because of the higher number of classes we generate, however we discovered that it is not necessarily the case.

Test	Specialized (stdev)	Generic (stdev)
funs	1.12 (0.02)	1.21 (0.06)
arrays	1.08 (0.03)	1.01 (0.02)

For the `funs` example, specialization may actually improve startup time, taking 1.12 times the average running time of the other iterations in the same VM invocation, compared to 1.21 in the generic case. We believe the improvement is due to not loading the boxed classes for integers. The second benchmark is more in line with our expectations, and the specialized version is slower, taking 1.08 times the average running times, compared to 1.01 for the generic case.

The results so far are very encouraging, and the startup time seems to be only slightly affected by the specialized transformation.

4. RELATED WORK

A number of languages use pervasive specialization for the implementation of generics. The best known example is template instantiation in C++ [2]. The drawbacks are risk of code explosion, lack of true separate compilation and a rigid notion of subtyping: instantiations of a template that are represented differently must have different types. Thus it would be impossible to have a subtyping relationship between `List[Int]` and `List[Any]`. Our approach lets the user decide what specializations to perform. Users can thus make a choice between performance and code size. We support separate compilation, and specialized instances are subtypes of the generic class, allowing for covariant subtyping.

At the virtual machine level, the most notable approach is taken by .NET generics [7]. Parameterized types are added to the VM intermediate language, which generates specialized implementations for classes on a by-need basis. The VM may choose to share code between implementations when data representation allows it. This alleviates the code size problem to some extent, and programs that instantiate generic types at primitive types run at full speed. In addition to performance, this approach makes full type information available at runtime. Our approach is different in that it needs no modifications to the virtual machine, it gives more control over what is specialized, and it allows for covariant subtyping.

In [6] the authors present a way to compile ML polymorphism that allows inspection of types at runtime. Programs can test the actual type and choose a more suitable representation. This is a very interesting approach and allows for more expressive programs, similar to C++ partial template specialization. However, it is not always possible to remove all type-level computation at compile time. Our approach does not need runtime representation of types.

Parametric types in Java are usually compiled through *type erasure* [3]. In [4] Cartwright and Steele present an approach based on specialization for full runtime type information. They generate a wrapper class and interface per type instantiation, the wrapper using an erasure-based implementation of the generic class. The wrapper encapsulates information about the specific instantiation and performs type tests and class instantiations. Our approach is similar in using specialized instances, but instead of generating them at the use-site, we do it at the definition. In addition, we focus on performance and rewrite code opportunistically to use specialized representations whenever possible.

5. FUTURE WORK

The compilation scheme we have presented so far has a limitation when a superclass of a generic class is instantiated at a specialized type. Consider the following example:

```
class List[@specialized A] {
..
}
class SpecialList[@specialized A] extends List[A] {
..
}
```

Each class generates a specialized subclass, `SpecialListInt` and `ListInt`. The problem becomes apparent when we ask what should be the superclass of `SpecialListInt`: it has to be the generic `SpecialList` (to allow mixing specialized and unspecialized code). But the superclass of `SpecialList` is `List`, not `ListInt`, therefore `SpecialListInt`, uses the generic representation of `List`. That means all inherited members are generic. While this does not break anything, it is not what a programmer would expect.

This limitation stems from our use of overriding for rerouting existing methods and field accesses. Whenever a specialized instance is created, the original methods are overridden to use the specialized representation. However, there is one thing that cannot be overridden on the Java Virtual Machine: the inheritance relationship. We believe a solution using multiple inheritance (mix-in composition in Scala) will work: whenever a specialized type parameter is used in a su-

pertype of a specialized class, we mix-in again that type in a specialized subclass. In the above example, `SpecialListInt` becomes:

```
class SpecialListInt
  extends SpecialList[Int] with ListInt {
..
}
```

The current implementation can be improved by allowing the programmer to specify at what types a type parameter should be specialized. It may be the case that specializing at all primitive types is wasteful, so we propose to parameterize the annotation at the specific types that need specialization. This can be as simple as a string argument, like `@specialized("Int")`.

Having specialized instances for primitive types could be used for giving more precise types at runtime. One could envision extending the transformation for more precise types at instanceof tests.

Another direction of improvement is user-defined specializations. Suppose we have a generic implementation of sets. While specialization on primitive types may already improve performance, the programmer may decide that a bit set representation is desirable. We could allow the `@specialized` annotation on classes and use the class given by the user instead of the one automatically generated by the compiler.

6. REFERENCES

[1] Bank, J. A., Myers, A. C., and Liskov, B. Parameterized types for Java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 132–145.

[2] Bjarne, S. *The C++ Programming Language.* Addison-Wesley, 1987.

[3] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM, pp. 183–200.

[4] Cartwright, R., and Steele, Jr., G. L. Compatible genericity with run-time types for the Java programming language. *SIGPLAN Not. 33*, 10 (1998), 201–215.

[5] Click, C., and Rose, J. Fast subtype checking in the HotSpot JVM. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande* (New York, NY, USA, 2002), ACM, pp. 96–107.

[6] Harper, R., and Morrisett, G. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, 1995), pp. 130–141.

[7] Kennedy, A., and Syme, D. Design and implementation of generics for the .NET Common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM, pp. 1–12.

[8] Lindholm, T., and Yellin, F. *The Java(TM) Virtual Machine Specification (2nd Edition).* Prentice Hall PTR, April 1999.

[9] Oderksy, M., Spoon, L., and Venners, B. *Programming in Scala.* artima, 2008.