Generic Numeric Programming Through Specialized Type Classes

Erik Osheim
Azavea
eosheim@azavea.com

Abstract

We describe an ongoing effort to build a system of type classes that support fast, accurate, flexible and generic numeric programming in Scala. This work combines Scala's support for user-directed type specialization with previous work on numeric type classes. In principle, these allow one to create generic numeric algorithms without sacrificing the speed of a direct implementation. In practice, these performance gains make very specific demands of both the language and the user.

This paper is a case study: we will explain the problems faced, discuss our strategies, and provide benchmarking results. We will also discuss ways in which Scala could be improved to more easily accommodate this kind of work. Finally, we will present a simple compiler plug-in that can be used to increase performance in many cases.

Categories and Subject Descriptors D [*3*]: 3

General Terms Experimentation, Performance

Keywords scala, specialization, numeric, generic, type class

1. INTRODUCTION

This paper concerns an effort over the last year to create a flexible, powerful, and high-performance implementation of the numeric type classes. The work was inspired by an email from Andreas Flierl to the Scala mailing list [4], and relies on the specialization support introduced in Scala 2.8 [3].

There are two implementations being discussed. The first was an R&D effort undertaken by Erik Osheim with the support of Azavea [1]. This project culminated in November 2011 with a GitHub release along with a series of blog posts and some profiling results [13]. The second project, Spire [14], is a general-purpose numerics library by Erik Osheim and Tom Switzer. Started as a set of proposed improvements to Scala's built-in numerics, Spire has evolved into a standalone library supporting new number types, a full type class hierarchy, and other functions.

Much of the underlying specialization work from the R&D project has been ported over to Spire, but some parts are only available in the original project (e.g. the compiler plug-in). Spire's number types and design philosophy have informed the design of its type classes, whereas the earlier project stayed closer to the design found in `scala.math`.

ScalaDays 2012 April 2012, London

2. BACKGROUND

2.1 Motivating Examples

Programming is often an exercise in abstraction. Developers value principles like "DRY" ("Don't repeat yourself") as well as design patterns like those described by the Gang of Four [5].

Generic numeric programming refers to implementations of algorithms and data structures that can be used with different underlying numeric types. One example of this is the `sum` method provided by many of Scala's collections, which adds the elements of the collection together and returns the total. Another example would be a matrix implementation that could be used with any numeric type (e.g. `Int`, `Double` or `BigDecimal`):

For example, suppose the distance formula is implemented as follows:

```
def hypot(x:Int, y:Int):Int = {
 val x2 = math.pow(x, 2)
 val y2 = math.pow(y, 2)
 math.sqrt(x2 + y2).toInt
}
```

This implementation is not very flexible: we'd need to create overloads to support any other numeric types we require (e.g. `Long`). Defining things in terms of `Double` seems to offer a way out, but it's not hard to find number types which `Double` can't accommodate: `BigDecimal` comes to mind. If we introduce a new number type (e.g. complex numbers), it's clear that this kind of overloading is not feasible in general.

Numeric type classes provide a more general solution to this problem. This paper will discuss the design of numeric type classes, their specific encoding in Scala, and attempts to mitigate the performance impact of writing generic numeric code.

2.2 Primitive Types and Reference Types

Java (as well as many other languages) distinguishes primitive types like `int` from reference types which extend `java.lang.Object`. When primitive types are passed as parameters or held in local variables, the values are stored directly in stack frames. Conversely, values of reference types refer to heap-allocated objects managed by the garbage collector. Objects also provide more flexibility than primitive types and can be used with Java's generics.

However, allocating objects is usually much more expensive than allocating primitive values, both in terms of memory overhead and also in terms of the performance of the garbage collector. In some cases the JVM will use escape analysis to allocate objects on the stack instead of the heap, but even these allocations have a cost [7].

2.3 Boxing and Unboxing

Because developers often want to abstract across primitive types Java provides primitive wrapper classes. For instance, a `java.lang.Integer` can be used to wrap an `int` value, a process is known as "boxing". When the `int` value is required, it can be retrieve via a call to `intValue` which is known as "unboxing".

```
// box two int values into Pair<Integer>
Integer a = new Integer(3);
Integer b = new Integer(4);
Pair<Integer> pair = new Pair(a, b);

// unbox the second value
int x = pair.second.intValue();
```

Boxing allows Java developers to abstract across primitive types, albeit with a cost both in performance (e.g. extra object allocations) and in expressiveness (calls to `new Integer()` and `intValue`). Scala does not have this distinction at the language level:

```
val a: Int = 3
val b: Int = 4
val pair: Pair[Int] = Pair(a, b)

val x: Int = pair._2
```

Scala unifies each primitive type (e.g. `int` in Java) with its boxed equivalent (`java.lang.Integer`) into a single type (`Int`). Unlike Java, where type parameters must be bound to reference types (Section 4.4 of [6]), generic methods in Scala may contain type parameters bound to primitive types. Java requires developers to be clear about the difference between the boxed and unboxed representation, whereas Scala doesn't require developers to worry about these distinctions. [10]

Despite these improvements, using `AnyVal` types in generic Scala functions still incurs the same performance penalty, albeit less explicitly. This is because the JVM requires all generic classes and methods to be parameterized on a reference type (section 4.3.4 of [9]). Scala uses the boxed representation when binding a primitive type to a type parameter.

2.4 Generic Math Operations in Scala

Number types in Scala lack any superclass or interface exposing arithmetic operations. This arrangement is inherited from Java but also reflects in the fact that primitive math operations are not implemented as methods in the JVM. Whatever the reason, types like `Int` and `Double` do not offer methods substantially different than the equivalent Java operations on primitives. Scala also offers functions like `pow(Double, Double)` in the scala.math package, which provides similar functionality to `java.lang.Math`.

Before 2.8, Scala did not provide any mechanism for generic numeric programming. In some cases `Double` acted as a stand-in, since the other numeric `AnyVal` types have implicit conversions to `Double` which mirror the coercions Java will perform. This can be useful but creates problems:

```
// inferred type of a is "Double"
val a = math.pow(5L, 3L)

val b = Long.MaxValue - 1

// imprecision causes failed assertion
assert(b == math.pow(b, 1).toLong)
```

While there are reasons why one might choose not to provide an implementation of `pow` for `Long` (namely that it would have O(n) time complexity as opposed to O(1) for `Double`) this example shows why `Double` is inappropriate as a generic numeric type.

This problem is even worse in the case of `BigInt` and `BigDecimal` whose unlimited ranges exceed `Double.MaxValue`. Instead of trying to find a single particular numeric representation, we can solve this problem using numeric type classes.

3. NUMERIC TYPE CLASSES

Type classes are a construction introduced by the Haskell programming language [15] to support ad-hoc polymorphism [18]. A type class can be loosely defined as a generic specification together with a set of types who can provide implementations of that specification parameterized on themselves. Unlike the classic inheritance model, members are not required to implement an interface via subtyping, but rather, provide instances which satisfy the type class' specification. [1]

Member types are linked to type classes via instances of the type class, parameterized on the member type. For in-

[1] General information about type classes is available at `http://www.haskell.org/tutorial/classes.html`.

stance, a type class called `Integral` might have an implementation in terms of `Int`. This means that there exists an instance of `Integral[Int]` containing the implementation of `Integral` in terms of `Int`.

In Scala, the type class pattern can be encoded using implicit values, implicit parameters and implicit conversions [12]. Particular instances of the type class are found via implicit search, passed to a methods as implicit parameters, and used to enrich generic types with the type class' supported operations.

One of the advantages of this pattern is that it can be used with types that don't share a common interface (either a supertrait or a superclass). The `AnyVal` types present just such a situation: they do not share a trait or superclass that provides useful functionality, although in practice they do share many useful properties. In principle type class instances can also be implemented by users to support types of which the type class itself has no knowledge.

3.1 Existing Type Classes in the Scala Library

The two most notable examples in the library are `Ordering`, which generalizes comparisons, and `Integral`, `Fractional`, and `Numeric` which generalize mathematical operations. Here's an example of a method written using Ordering:

```
import scala.math.Ordering._
import scala.math.Ordering.Implicits._

// scala as written
def lessThan[A:Ordering](x:A, y:A) = x < y

// with syntactic sugar removed
def lessThan[A](x:A, y:A)(ev:Ordering[A]) = {
  x.<(y)
}

// after implicits are resolved
def lessThan[A](x:A, y:A)(ev:Ordering[A]) = {
  ev.infixOrderingOps(x).<(y)
}

// with ev.infixOrderingOps inlined
def lessThan[A](x:A, y:A)(ev:Ordering[A]) = {
  new ev.Ops(x).<(y)
}
```

Type classes like `Ordering` enable Scala to reuse a single method implementation for many desired data type. This keeps the size of the library low (in terms of lines of code), and require less coupling between library interfaces and user-defined types. For example, Scala's collections implement `sum`, `min` and `max` in terms of `Numeric` and `Ordering`.

3.1.1 Performance Costs

Unfortunately these type classes do come with a performance cost. Benchmarks conducted around `Ordering` suggest that using a generic implementation with Long is 3.3-12.8 times slower than a direct implementation that uses Long directly (see Appendix C).

In fact, `scala.util.Sorting#quickSort` provides direct implementations for `Array[Int]`, `Array[Float]` and `Array[Double]` precisely to avoid this kind of performance penalty. When used with `Array[Long]` we find that `quickSort` (using the Ordering-based implementation) is 4-6 times slower than the other direct implementations (as shown in Appendix B).

3.1.2 Expressiveness

Not only do the generic methods result in worse performance, but in many cases they don't define all the operations on the generic type which users need. For instance, `scala.math.Numeric` (which abstracts across all the numeric types, like `Int` and `Double`) does not support division. This means that many algorithms can't be generalized across integral and fractional types, even when the division in question would be well-defined (see `https://issues.scala-lang.org/browse/SI-4658`).

The `Fractional` type class (which generalizes types like `Float`, `Double` and `BigDecimal`) does support division. Even so, it has many similar problems. For one, `Fractional` doesn't contain logarithms, exponentiation or trigonometry functions. To use those one must convert to `Double` (which will not work for all `BigDecimal` values). In fact, there is no support for converting to (or from) `BigDecimal` at all, which means that `Fractional` is effectively limited to double precision. Worse, `Numeric`, `Fractional` and `Integral` all extend `Ordering`, which means they can't be extended by users to include types like complex numbers.

3.2 Goals and Strategy

Users have different ideas of what generic numeric support should entail. Some users will want a type class which is maximally flexible (supporting division, exponentiation, ordering, etc.) even when some number types may not fully support the operation. This could mean that a type uses an approximation (for instance, an integral type might implement `/` in terms of floor division) or that an operation is not supported (for instance, a complex number might throw an exception when used with `<`).

Other users will want to structure type classes algebraically, either to make guarantees related to precision or to enforce greater type safety. Naturally, there is tension between performance, flexibility, and precision. Rather than trying to find a single strategy to try to please all users of Spire, we have chosen to support two different type class approaches.

The first approach consists of the `Numeric` type class. It supports all mathematical operations on all number types with the possibility of loss of precision or runtime failures– its goals are speed and flexibility. This corresponds to the strategy taken in the original R&D project.

The second approach defines a tower of type classes, each level offering more and more capabilities to a restricted set

of numeric types. Using these type classes an author can precisely define the algebraic properties required of a type. Its goals are speed and type safety (i.e. guarantees about a type's properties). The exact structure of these type classes is discussed in Section 5.2.

Besides structuring the numeric type classes to gain flexibility and support future numeric types, a goal of Spire is also important to maintain performance on par with direct implementations. If direct implementations are known to be several orders of magnitude faster than generic, developers will avoid supporting all number types (a situation we find ourselves in today). The JVM is able to optimize much of the type class indirection away, but to realize our performance goals we will need to use specialization.

4. SPECIALIZATION

The main cause of the decline in performance is boxing: when an `AnyVal` type is used as a type parameter it must be used in the boxed form (since generic type parameters are converted to `Object` at the JVM level). Fortunately, we can often avoid this unnecessary boxing by using specialization, a feature introduced to Scala in 2.8 to solve exactly these kinds of problems [3].

Specialization allows us to create direct, "specialized" versions of particular classes or methods. When a specialized variant is available, the compiler will prefer it to the generic implementation. For example, if our previous example was specialized on `Int`, the compiler would generate the following prototypes:

```
// method as written
def lessThan[@spec(Int) A:Ordering](a:A, b:A):
  Boolean

// compiles to generic version
def lessThan(a:Object, b:Object, ev:Ordering):
  Boolean

// version specialized on Int, without boxing
def lessThan$mIc$sp(a:Int, b:Int, ev:Ordering):
  Boolean
```

4.1 Current Limitations of Specialization

Specialized type parameters necessarily increases the amount of bytecode generated (by roughly 2-10 times per class, depending upon how many types are specialized). This increase becomes a combinatorial explosion when there are multiple specialized type parameters (since each possible combination generates a unique specialized implementation). Thus, there is an explicit trade-off between code size, performance and features.

It's important to remember that specialization currently places restrictions on how inheritance can be used. Subclasses of a specialized parent class will inherit from (and thus use) the parent's generic implementation, even if a specialized version exists and would be preferred. Since extending a specialized trait does use the parent's specialized implementation, we prefer to use traits rather than abstract classes. This restriction may change in future versions of Scala.

Finally, there some cases where specialization doesn't perform as expected. For instance, specialized classes with methods that introduce additional specialized type parameters often do not avoid boxing. Here is a (simplified) example:

```
import scala.{specialized => sp}

object Baz {
 def baz[@sp(Int) A, @sp(Int) B](a:A, b:B):String =
  ...
}

trait Foo[@sp(Int) A] {
 val a:A
 def bar[@sp(Int) B](b:B) = Baz.baz(a, b)
}
```

We would like for `new Foo(3).bar(4)` to generate calls to `Baz.baz$mIIc$sp` (the specialized version of `Baz.baz`) without any intermediate boxing. Cases where `A` or `B` are not set to `Int` would generate calls to `Baz.baz` with two instances of `Object`. While this example is somewhat contrived it will become more important in section 5.3 when we discuss generic numeric conversions.

Here is a Scala-like representation of what the compiler actually produces. For each trait, Scala produces an interface and an abstract class with the implementation.

```
object Baz {
 // generic version of "baz"
 def baz(a:Object, b:Object):String = ...

 // specialized version of "baz"
 def baz$mIIc$sp(a:Int, b:Int):String = ...
}

// generic Foo interface
interface Foo extends ScalaObject {
 def a():Object
 def a$mcI$sp():Int
 def bar(b: Object):String
 def bar$mIc$sp(b: Int):String
}

// generic Foo implementation
abstract class Foo$class extends Object {
 def a$mcI$sp($this:Foo) = unbox($this.a())

 def bar($this:Foo, b:Object) = {
  Util.baz($this.a(), b)
 }

 def bar$mIc$sp($this:Foo, b:Int) = {
  Util.baz($this.a(), scala.Int.box(b))
 }
}

// specialized Foo[Int] interface
interface Foo$mcI$sp extends Foo {
 def a():Int
}
```

```
// specialized Foo[Int] implementation
abstract class Foo$mcI$sp$class extends Object {}
```

Leaving aside the exact details of why this class structure is generated, notice that both versions of `bar` which were generated will result in boxing, since neither of them calls `baz$mIIc$sp`. A similar thing happens when using specialization with inner and outer classes.

This is probably the biggest limitation of specialization: the library author really has to dig in and understand what code is being generated, since the interactions are often murkier than they would appear. This will hopefully become less of a concern as the feature matures.

4.2 Specialized Type Classes

Type class traits in Scala are perfectly amenable to specialization, keeping in mind the previous caveats. Unfortunately, since the type classes in the standard library are not specialized, inter-operating with or extending these types negates the advantages that specialization gains us. More generally, we need to make sure we are using specialized classes and methods from the call site all the way down to the direct type class implementation.

Also, even without boxing primitive values, the type enrichment mechanism does allocate a new object (e.g. a `NumericOps` instances) every time an infix operator is used. Escape analysis enables the JVM to allocate these objects on the stack, but there is still a performance cost. The cost is less severe than boxing but if the goal is parity with direct implementations then it becomes a factor. This cost can be dodged by calling `Numeric[A]`'s methods explicitly, but the resulting syntactic noise is quite bad:

```
def clean[A:Numeric](x:A, y:A) = x * x + y

def fast[A](x:A, y:A)(implicit ev:Numeric[A]) =
 ev.plus(ev.times(x, x), y)
```

5. IMPLEMENTATION

With the previous points in mind, we must provide an independent implementation of all the relevant type classes, from equivalence relations and ordering up through the mathematical operations, in order to receive the benefits of specialization. We will also structure our type classes to avoid things like inner classes, anonymous classes and other constructions which interfere with specialization.

This section will discuss the design of Spire.

5.1 Numeric Type Class

The first strategy, the `Numeric` type class, is intended as a general purpose "number abstraction" which supports all the operations defined on primitives. Since operators like `/` and `%` are defined on both `Int` and `Double`, `Numeric` defines these operators, with the understanding that the meaning of `a / b` is contextual on which `Numeric[A]` is being used.

Type members of `Numeric` are required to implement a wide range of operators and to do the job as best they can. This means that quotient and division will both be defined in terms of `Int#/` since integer types often can't represent the results of division. A complex number type implementing `Numeric` would need to throw runtime errors when comparison operators were used. And some operations might overflow or result in a loss of precision. `Numeric` makes a purposeful trade-off, sacrificing some numerical soundness in exchange for flexibility.

5.2 Ring, EuclideanRing, Field Type Classes

The second strategy encompasses a family of type classes which mirror the library's current type classes, but attempts to stay closer to the underlying algebras. The hierarchy consists of:

- Eq: `===, =!=` (typed equality)
- Order: Eq with `<, >`
- Ring: Eq with `+, -, *, pow, abs`
- EuclideanRing: Ring with `quot, mod`.
- Field: EuclideanRing with `/`.
- Integral: EuclideanRing with Order.
- Fractional: Field with Order.

The most basic of these is `Ring` which corresponds to an algebraic ring with unity and is similar to Haskell's `Num`. Each additional type class extends (and builds on) the supported operations. Both ordered and unordered type classes are available. Type members of these type classes are held to a much stricter standard than with `Numeric` and fully support the interfaces in question.

While the overall approach is inspired by Haskell's, there are some important differences. For one, we have resisted creating type classes specifically for floating-point implementations (Haskell defines `Floating` and `RealFloat`). For another, we have allowed `Field` to inherit quotient and remainder from `EuclideanRing`–there didn't seem to be any good reason to restrict those to be available only for integral types. Finally, we have supported more operations than Haskell does: `Ring` implements `pow`, and we support trigonometry and `nroot` via their own type classes [2], rather than baking them into `Floating`.

5.3 Conversions Between Types

Once one starts working with the previously defined type classes, there are a few things that one quickly notices. One is how many methods are defined only for `Double`. But another is that using numeric constants (such as 42) in code becomes quite cumbersome. For example:

```
// direct implementation in terms of Int
def add42(a:Int) = a + 42

// generic implementation
```

[2] See the Trig and NRoot type classes in Spire.

```
def add42[A:Numeric](a:A) = {
  a + Numeric[A].fromInt(42)
}
```

This is not a show-stopper but it does reduce the readability of generic methods a bit. This gets worse as you find yourself converting from your generic type to other types and back again:

```
// direct implementation in terms of Int
def hypot(x:Int, y:Int):Int = {
 val x2 = math.pow(x, 2)
 val y2 = math.pow(y, 2)
 math.sqrt(x2 + y2).toInt
}

// generic implementation
def hypot[A:Numeric](x:A, y:A):A = {
 val n = numeric
 val x2 = x.pow(2)
 val y2 = y.pow(2)
 n.fromDouble(math.sqrt(n.toDouble(x2 + y2)))
}
```

One solution which was explored in [1] was to define a set of implicit operators which would automatically convert to the correct type. Here is a simplified example:

```
class Ops[@spec A:Numeric](lhs:A) {
  val n = implicitly[Numeric[A]]
  def +[B:ConvertableFrom](rhs:B) = {
    n.minus(lhs, n.fromType(rhs))
  }
}

// implementation using ConvertableFrom
def add42[A:Numeric](a:A) = a + 42
```

`Numeric[A]#fromType` uses the `ConvertableFrom[B]` instance to convert from `B` to `A` before performing the addition. This strategy allows us to mix literals, concrete types, and generic types with generic numeric types.

There are some disadvantages to this approach. First, the `+` operator in particular will generate an ambiguous implicits error because of the `Predef.any2stringadd` implicit. This can be fixed by masking `any2stringadd` but this must be done in every file that uses these implicits, which creates boilerplate.

Worse, due to the specialization bug described earlier, these methods introduce boxing even when specialized, which degrades performance. This is in addition to the performance penalty introduced by type enrichment. Without recourse to compiler plug-ins or changes to specialization, this approach is not feasible.

5.4 Type Class Extensibility

This type class hierarchy was designed to support user-defined number types beyond those contained in Scala. Examples of this from [14] include `Complex[A]` and `Rational`.

The only place where the types are not fully extensible are numeric conversions. The current strategy defines things like `toInt` and `fromInt` in the interface itself, so future number types (like a hypothetical `Double128`) would not be supported by the same range of automatic conversions. It's possible that a more extensible strategy for managing conversions without sacrificing performance could be found.

5.5 Optimizing Compiler Plug-in

One of the final features which drastically improved performance was the creation of a compiler plug-in for removing intermediate object creation during type enrichment [1]. This plug-in transforms a valid Scala program using type enrichment into one that calls methods explicitly on the implicit type class instance. For example:

```
// using type enrichment
def foo[@spec A:Numeric](a:A, b:B) = a + b

// what type enrichment expands to
def bar[@spec A](a:A, b:B)(implicit ev:Numeric[A]) = {
 // allocates a NumericOps instance
 new NumericOps(a).+(b)
}

// what the compiler plug-in creates
def baz[@spec A](a:A, b:B)(implicit ev:Numeric[A]) = {
 ev.plus(a, b)
}
```

This allows us to benefit from the clean syntax of type enrichment without having to allocate Ops instances every time we use an operator. This is a key feature: otherwise authors will be conflicted between writing clean, expressive code and code that runs quickly.

The compiler plug-in used in [1] has a hard-coded mapping from enriched methods to methods on the implicit `Numeric[A]` parameter. Future plug-ins should be able to automatically do this mapping based on an annotation.

6. RESULTS

6.1 Performance Tests

The micro-benchmark results are extremely positive. With the optimizing plug-in, most generic benchmarks run within +/- 10% of their direct equivalents. More importantly, none of generic benchmarks run more than 40% slower. By comparison, the "old" generic benchmarks tend to be about 300-700% slower, and in some cases are hundreds of times slower than the comparable direct benchmark. Without the optimizing compiler plug-in, writing generic code using infix notation is somewhat slower, but is still 2-3x faster than the "old" generic type classes.

When defining operations in terms of the `ConvertableFrom` type class the infix notation gets even slower. Calling methods on the implicit parameter restores the performance in some cases, but at the cost of clunky syntax, and other cases are always slow due to the quirks of specialization. Without a compiler plug-in or more precise specialization support this strategy is not feasible.

Appendix B has a detailed table of results, as well as a discussion of the benchmarking methodology.

6.2 Syntax Quirks

6.2.1 Promotion and precision

First, it's worth noticing that methods implemented in terms of numeric type classes have no way to compare the precision of a generic type parameter with other concrete types. One could imagine the following:

```
// return Double or A, whichever is wider
def timesPi[A:Fractional](a:A):Double or A = {
  a * Double.PI
}
```

This is not practical for a few reasons. First, the return type would have to be dynamically decided at run time, complicating the type signature. Second, introducing new type parameters only in the return type will require users to be explicit about the return type in ways they won't expect. Thus, the generic parameter in question (`A`) is set for a particular type, and can't be promoted.

6.2.2 Operator mishaps

In addition to the previously-mentioned problems with the `any2stringadd` implicit there are some other gotchas around operators.

To support the distinction between floor division (defined on `EuclideanRing`) and normal division (defined on `Field`) we introduced `/~` as a floor division operator, along with `/%` as a divmod operator. It would have been possible to use a method name like `quot` instead, but `/~` has the advantage of having correct precedence for division.

The `pow` operator presented problems as well. An operator with correct precedence would need to start with "~" or similar, and to associate correctly would need to end with ":". However, the prospect of using `~^:` for exponentiation was not attractive. Spire currently uses two exponentiation operators: `pow` and `**`.

7. FUTURE WORK

This section will discuss future work to be explored in Spire.

7.1 Improving/specializing type classes

These results suggest that there is still much low-hanging fruit related to optimizing the type class pattern with specialization. Even without using a compiler plug-in a dramatic speed up is currently possible, although some restructuring is required to realize these potential gains. Libraries like Scalaz [16] which make heavy use of type classes could benefit from specialization.

The specialization feature itself could also be improved. Many of the constraints around structuring type classes are to work around the nuances of the current specialization scheme. As specialization improves these guidelines will change, or disappear.

7.2 Boxed number type

There are some features which do not interact well with a type class approach. One example is detecting possible overflow and promoting the result type to avoid it (e.g. by promoting an `Int` to a `Long`). Other examples include precision tracking, as well as supporting operations on mixed numeric types intelligently (e.g. promoting the result of BigInt + Double to BigDecimal).

While it seems odd to suggest adding a boxed number type after railing against the performance of boxing there are some situations where performance is less important than flexibility and correctness. Boxed numbers would enable the implementation of a numeric tower like those found in Scheme [8] and Python [19].

7.3 Optimizing compilation

Given, the excellent performance boost it provides, porting the compiler plug-in to Spire is a high priority. Unfortunately given the much larger number of methods and classes it will be significant work to port the prototype's current design.

The compiler plug-in could be generalized to support arbitrary classes through a user-annotation, somewhat like the current `@inline` annotation. The current design is brittle due to hard-coding the operator names and implementation details–a better version would find the wrapper class automatically (potentially through an annotation), detect implicit conversions to that class and inline the class' method body directly.

Finally, the kinds of optimizations the compiler plug-in performs could easily be folded into the Scala compiler. This would benefit not just specialized type classes but all uses of enrichment. There has been some discussion around "inlined implicit classes" and Josh Suereth's implicit class SIP [17]. Value classes [11] and macros [2] may also be able to help here.

A. PROJECT SOURCE CODE

This paper refers to source code taken from two different projects to specialize the numeric type classes.

The source code for the original specialized numeric R&D project is located at: `https://github.com/azavea/numeric`. This project is historical and not actively developed.

The source code for Spire is located at: `https://github.com/non/spire`. Spire is under development and moving towards a first release.

B. PERFORMANCE TEST RESULTS

These are the performance numbers generated without using the compiler plug-in. The infix-adder tests as well as increment-int 2-4 show degraded performance due to the large number of intermediate objects created. The other tests have been written in a style which avoids doing this, to demonstrate the potential performance improvements.

Numeric Tests Results (ms)			
test	direct	new	old
infix-adder-int	1.3	7.9	14.1
infix-adder-long	1.9	8.0	21.0
infix-adder-float	3.5	5.9	16.3
infix-adder-double	3.6	4.5	16.8
array-total-int	1.5	1.4	21.3
array-total-long	1.9	1.1	20.9
array-total-float	3.6	2.8	17.0
array-total-double	3.6	3.3	17.8
find-max-int	2.1	3.4	17.8
find-max-long	2.1	1.6	19.1
find-max-float	6.0	4.4	18.5
find-max-double	5.0	5.4	16.8
insertion-sort-int	1.3	1.9	13.4
insertion-sort-long	0.9	1.3	14.8
insertion-sort-float	2.0	1.8	19.0
insertion-sort-double	1.1	0.8	13.4
increment-int1	0.1	0.1	13.4
increment-int2	0.1	3.5	11.9
increment-int3	0.1	4.5	12.5
increment-int4	0.1	5.4	11.9

The optimizing compiler plug-in was built against Scala 2.9.1. It used exactly the same benchmarking code (and the same libraries) as the above tests; the only difference is the additional `scalac` argument to enable the compiler plug-in.

Numeric With Optimizing Plug-in (ms)			
test	direct	new	old
infix-adder-int	1.1	1.6	14.9
infix-adder-long	1.5	1.5	20.9
infix-adder-float	3.9	4.1	19.5
infix-adder-double	3.0	3.4	18.5
array-total-int	0.9	1.5	21.1
array-total-long	2.0	1.8	21.0
array-total-float	3.6	2.8	18.4
array-total-double	3.3	3.6	20.5
find-max-int	4.0	4.0	17.5
find-max-long	1.8	1.9	19.3
find-max-float	4.9	5.8	16.9
find-max-double	5.5	3.8	16.5
insertion-sort-int	1.4	1.1	14.1
insertion-sort-long	1.4	0.9	15.1
insertion-sort-float	1.3	1.6	17.6
insertion-sort-double	1.9	2.0	14.6
increment-int1	0.1	0.1	11.9
increment-int2	0.1	0.4	12.1
increment-int3	0.1	0.1	12.3
increment-int4	0.1	0.3	12.9

The actual benchmarking code used in these tests is available at [1], in the "perf" subproject. In addition to the benchmarks named in the paper, the project contains others which were not used. Some were omitted because they were not considered relevant (e.g. the tests using the library version of quickSort/Ordering) others duplicated very similar tests (and showed similar results).

All implementations use arrays and while loops to maximize the effect of the arithmetic operations themselves. The "new" column corresponds to the specialized type classes, the "old" to the type classes in the Scala library now, and "direct" corresponds to an implementation written in terms of the particular type in question.

These tests were all run in Scala-2.9.1 and Java 1.6.0_26 using the `-optimize` flag, on a Core i7 processor with 8G of RAM.

C. OTHER BENCHMARKING RESULTS

These tests were run to compare direct implementations of basic ordering methods with generic implementations using `scala.math.Ordering`:

(time in ms)	Generic	Direct	Slowdown
LessThan	46.00	13.87	3.32x
EqualTo	74.41	5.86	12.70x
FindMax	71.57	5.84	12.26x

The code for this benchmark is available at `https://github.com/non/ordering-benchmark`.

Like the previous benchmarks, these tests were run in Scala-2.9.1 and Java 1.6.0_26 using the `-optimize` flag, on a Core i7 processor with 8G of RAM.

Acknowledgments

I would like to acknowledge the research and development time that Azavea has generously granted without which this paper would not exist.

Many thanks to Tom Switzer for invaluable discussions, as well as for his work on Spire. It has been a pleasure to collaborate with him. I would also like to acknowledge others whose interest and work in specializing numeric preceded mine, particularly Iulian Dragos and Andreas Flierl. Finally, thanks to the Scala community for help, support and feedback.

References

[1] AZAVEA, 2011. “com.azavea.numeric” https://github.com/azavea/numeric.

[2] BURMAKO, Eugene, ODERSKY, Martin, VOGT, Christopher, ZEIGER, Stefan and MOORS, Adriaan. 2012. “Self-cleaning macros” http://scalamacros.org/documentation.html

[3] DRAGOS, Iulian and ODERSKY, Martin. 2009. “Compiling Generics Through User-Directed Type Specialization” http://lamp.epfl.ch/~dragos/files/scala-spec.pdf

[4] FLIERL, Andreas 2011. “specializing Numeric” http://scala-programming-language.1934581.n4.nabble.com/specializing-Numeric-td3171386.html

[5] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph and VLISSIDES, John. Addison-Wesley 1994. “Design Patterns: Elements of Reusable Object-Oriented Software”

[6] GOSLING, James, JOY, Bill, STEELE, Guy, BRACHA, Gilad and BUCKLEY, Alex. 2012. “The Java Language Specification: Java SE 7 Edition” http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

[7] JUMA, Ismael. 2008. “Objects with no allocation overhead” http://blog.juma.me.uk/2008/12/17/objects-with-no-allocation-overhead/

[8] KELSEY, Richard, CLINGER, William and REES, Jonathan eds. 1998. “Revised^5 Report on the Algorithmic Language Scheme”

[9] LINDHOLM, Tim, YELLIN, Frank, BRACHA, Gilad and BUCKLEY, Alex. 2011. “The Java Virtual Machine Specification: Java SE 7 Edition” http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[10] ODERSKY, Martin, SPOON, Lex and VENNERS, Bill. Artima 2011. “Programming in Scala”

[11] ODERSKY, Martin, OLSON, Jeff, PHILLIPS, Paul and SUERETH, Joshua. 2012. “SIP 15: Value Classes” http://docs.scala-lang.org/sips/pending/value-classes.html

[12] OLIVEIRA, Bruno C. d. S., MOORS, Adriaan and ODERSKY, Martin. 2010. “Type classes as objects and implicits”

[13] OSHEIM, Erik 2011. “Scala’s Numeric type class” http://www.azavea.com/blogs/labs/2011/06/scalas-numeric-type-class-pt-1/

[14] OSHEIM, Erik and SWITZER, Tom. 2011. “Spire” https://github.com/non/spire.

[15] PEYTON-JONES, Simon, ed. 2002. “Haskell 98 Languages and Libraries: The Revised Report” http://www.haskell.org/onlinereport/basic.html

[16] SCALAZ. 2008-2011. https://github.com/scalaz/scalaz

[17] SUERETH, Josh. 2011. “SIP 13: Implicit Classes” http://docs.scala-lang.org/sips/pending/implicit-classes.html

[18] WADLER, Philip and BLOTT, Stephen. 1988. “How to make ad-hoc polymorphism less ad hoc”

[19] YASSKIN, Jeffrey. 2010. “PEP 3141: A Type Hierarchy for Numbers” http://www.python.org/dev/peps/pep-3141/