Implicit conversions in the Scala Algebra System (extended abstract)

Raphaël Jolly

Databeans, Vélizy-Villacoublay, France, `raphael.jolly@free.fr`

1 Introduction

The Scala Algebra System [1] is a computer algebra project written in Scala. The Scala programming language allows to conveniently define algebraic categories with its type classes and extension methods [2, 3]. It also provides implicit conversions, but these turn out to be difficult to operate in conjunction with said features. Moreover, discussions are on-going about restricting their operation, which is currently too heavyweight [4]. This complicates the subject even further, but might also bring new opportunities.

2 Enrichment and promotion

One of the most important features a general purpose language has to provide for computer algebra is custom types, and among these a type for multiprecision arithmetic. Object oriented languages are well suited for custom types, and this is the case of Java. Its multiprecision arithmertic type is `java.math.BigInteger`. Its syntax however is verbose and two more features will have to be found elsewhere : *enrichment* and *numeric promotion*. Enrichement allows to endow a numeric type with arithmetic operators, while promotion allows to lift an operand value from a subset type to the type of the other operand.

In Scala there are two possible approaches for enrichement : *wrapper* and *typeclass*. In the standard library, a multiprecision arithmetic type `BigInt` is provided, which is of the wrapper kind. Numeric promotion for this type is supported, as shown below:

```
val a = BigInt(1)
val b = a + 1
val c = 1 + a
```

In the typeclass approach however, promotion on the left operand does not work:

```
import java.math.BigInteger
import scala.language.implicitConversions

trait Ring[T]:
  extension (x: T) def + (y: T): T

given r: Ring[BigInteger] with
  given Conversion[Int, BigInteger] = BigInteger.valueOf( )
  extension (x: BigInteger) def + (y: BigInteger) = x.add(y)

import r.given

val a = BigInteger("1")
val b = a + 1 // works
val c = 1 + a // fails
```

3 Wrapper vs typeclass approach

In the wrapper approach numeric promotion is enabled by the definition below in object `BigInt`:

```
implicit def int2bigInt(i: Int): BigInt = apply(i)
```

The Scala specification states in which situations implicit conversions (or *views*) are applied. For promotion on the right operand it is:

> 1) If an expression `e` is of type `T`, and `T` does not conform to the expression's expected type `pt`. In this case, an implicit `v` which is applicable to `e` and whose result type conforms to `pt` is searched. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of `T => pt`. If such a view is found, the expression `e` is converted to `v(e)` [5]

In the example in Section 2, the type of the parameter to `+` is `Int` whereas its expected type is `BigInt`, so an implicit conversion is looked up in the implicit scope of `Int => BigInt`, which includes `BigInt`'s companion object.

For promotion on the left operand, the situation is:

> 3) In an application `e.m(args)` with `e` of type `T`, if the selector `m` denotes some accessible member(s) of `T`, but none of these members is applicable to the arguments `args`. In this case, a view `v` which is applicable to `e` and whose result contains a method `m` which is applicable to `args` is searched. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of `T => pt`, with `pt` being the structural type `{ def m(args: T 1 , ... , T n): U }`. If such a view is found, the application `e.m(args)` is converted to `v(e).m(args)`

So we have an explanation why promotion on the left does not work in the typeclass approach : the relevant type is Java's `BigInteger` and there is no implicit scope where to look for an implicit conversion. The problem can be circumscribed by importing the operator from the typeclass instance, which is done using a wildcard `*`:

```
import r.{given, *}
...
val c = 1 + a // works
```

4 Defining several instances in the same scope

One further problem occurs if we try to define more than one typeclass instance in the same scope. Promotion on the right operator still works, but not on the left:

```
case class MyInt1(n: Int)
case class MyInt2(n: Int)

given r: Ring[MyInt1] with
  given Conversion[Int, MyInt1] = MyInt1( )
  extension (x: MyInt1) def + (y: MyInt1) = MyInt1(x.n + y.n)

given s: Ring[MyInt2] with
  given Conversion[Int, MyInt2] = MyInt2( )
  extension (x: MyInt2) def + (y: MyInt2) = MyInt2(x.n + y.n)

import r.{given, *}
import s.{given, *}

MyInt1(1) + 1 // ok
MyInt2(1) + 1 // ok
1 + MyInt1(1) // None of the overloaded alternatives of method +
    in class Int with types ... match arguments (MyInt1)
1 + MyInt2(1) // None of the overloaded alternatives of method +
    in class Int with types ... match arguments (MyInt2)
```

Whe have a hint of what is wrong if we substitute a operator that is not already part of type `Int`:

```
1 ++ MyInt1(1)
^^^^
value ++ is not a member of Int.
An extension method was tried, but could not be fully constructed:

    s.++(s.given Conversion Int MyInt2.apply(1))

    failed with:

        Ambiguous extension methods:
        both s.++(s.given Conversion Int MyInt2.apply(1))
        and  r.++(r.given Conversion Int MyInt1.apply(1))
        are possible expansions of 1.++
```

The extension method resolution does not try hard enough and fails to see that one of the possible solutions is working better than the other. This limitation of the Scala compiler can be mitigated by some arrangements, which will be discussed in the presentation.

References

1. Jolly, R.: ScAS - Scala Algebra System. Tech. rep., https://github.com/rjolly/scas (2010-2024)
2. Jolly, R.: Categories as type classes in the Scala Algebra System. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC. Lecture Notes in Computer Science, vol. 8136, pp. 209–218. Springer (2013), https://link.springer.com/chapter/10.1007/978-3-319-02297-0 18
3. Jolly, R.: Progress report on the scala algebra system. In: Computer Algebra in Scientific Computing: 22nd International Workshop, CASC 2020, Linz, Austria, September 14–18, 2020, Proceedings 22. pp. 307–315. Springer (2020)
4. Scala contributors: Proposed changes and restrictions for implicit conversions. Tech. rep., https://contributors.scala-lang.org/t/proposed-changes-and-restrictions-for-implicit-conversions/4923 (2021)
5. Scala developers: Implicit conversions - more details. Tech. rep., https://docs.scala-lang.org/scala3/reference/changed-features/implicit-conversions-spec.html (2002-2024)