Ruby master - Feature #8223

Make Matrix more omnivorous.

04/05/2013 08:55 PM - Anonymous

Status:	Open	
Priority:	Normal	
Assignee:	marcandre (Marc-Andre Lafortune)	
Target version:		

Description

Let's imagine a class Metre, whose instances represent physical magnitudes in metres.

```
class Metre
  attr_reader :magnitude
  def initialize magnitude; @magnitude = magnitude end
  def to_s; magnitude.to_s + ".m" end
end
```

Let's say that metres can be multiplied by a number:

```
class Metre
  def * multiplicand
    case multiplicand
    when Numeric then Metre.new( magnitude * multiplicand )
    else
      raise "Metres can only be multiplied by numbers, multiplication by #{multiplicand.class} att
empted!"
    end
  end
end
```

And that they can be summed up with other magnitudes in metres, but, as a feature,
not with numbers (apples, pears, seconds, kelvins...).

```
class Metre
  def + summand
    case summand
    when Metre then Metre.new( magnitude + summand.magnitude )
    else
      raise "Metres can only be summed with metres, summation with #{summand.class} attempted!"
    end
  end
end
```

Now with one more convenience constructor Numeric#m:

```
class Numeric
  def m; Metre.new self end
end
```

We can write expressions such as

```
3.m + 5.m
#=> 8.m
3.m * 2
#=> 6.m
```

And with defined #coerce:

```
class Metre
  def coerce other; [ self, other ] end
end
```

Also this expression is valid:

```
2 * 3.m
#=> 6.m
```

Before long, the user will want to make a matrix of magnitudes:

```
require 'matrix'
mx = Matrix.build 2, 2 do 1.m end
#=> Matrix[[1.m, 1.m], [1.m, 1.m]]
```

It works, but the joy does not last long. The user will fail miserably if ze wants to perform matrix multiplication:

```
cv = Matrix.column_vector [1, 1]
mx * cv
#=> RuntimeError: Metres can only be summed with metres, summation with Fixnum attempted!
# where 2.m would be expected
```

In theory, everything should be O.K., since Metre class has both metre summation and multiplication by a number defined. The failure happens due to the internal workings of the Matrix class, which assumes that the elements can be summed together with numeric 0. But it is a feature of metres, that they are picky and allow themselves to be summed only with other Metre instances.

In my real physical units library that I have written, I have solved this problem by
defining an über zero object that produces the expected result, when summed with objects, that would otherwise not lend themselves to summation with ordinary numeric 0,
and patching the Matrix class so that it uses this über zero instead of the ordinary one.

But this is not a very systematic solution. Actually, I think that the Matrix class would be more flexible, if, instead of simply using 0, it asked the elements of the matrix what their zero is, as in:

```
class << Metre
  def zero; new 0 end
end
```

But of course, that would also require that ordinary numeric classes can tell what their zero is, as in:

```
def Integer.zero; 0 end
def Float.zero; 0.0 end
def Complex.zero; Complex 0.0, 0.0 end
# etc.
```

I think that this way of doing things (that is, having #zero methods in numeric classes and making Matrix actually require the class of the objects in it to have public class method #zero defined) would make everything more consistent and more algebra-like. I am having this problem for already almost half a year, but I only gathered courage today to encumber you guys with this proposal. Please don't judge me harshly for it. I have actually already seen something like this, in particular with bigdecimal's Jacobian (http://ruby-doc.org/stdlib-2.0/libdoc/bigdecimal/rdoc/Jacobian.html), which requires that the object from which the Jacobian is computed implements methods #zero, #one, #two etc. Sorry again.

History

#1 - 04/07/2013 04:30 AM - Anonymous

For starters, I propose changing the existing Matrix#* method to the following:

class Matrix
Matrix multiplication.
#
def * arg # arg is matrix or vector or number
case arg
when Numeric
rows = @rows.map { |row| row.map { |e| e * arg } }
return new_matrix rows, column_size
when Vector
arg = Matrix.column_vector arg
result = self * arg
return result.column 0
when Matrix
Matrix.Raise ErrDimensionMismatch if column_size != arg.row_size
rows = Array.new row_size do |i|
Array.new arg.column_size do |j|
(0...column_size).map { |c| arg[c, j] * self[i, c] }.reduce :+
end

end
return new_matrix(rows, arg.column_size)
else
compat_1, compat_2 = arg.coerce self
return compat_1 * compat_2
end
end
end

Provided that I didn't make a mistake, this version of matrix multiplication does not perform
addition of 0 to the sum. I know that there is actually a reason why we should always write
#reduce with starting value, but I now cannot remember what the reason was. And patching with
this method makes the matrix multiplication of metres work...

#2 - 04/07/2013 04:48 AM - Anonymous

Ahh, I remembered now, initial zero has to be provided to make empty collections work.
My suggested patch breaks empty matrix multiplication. I hope Marc-Andre will appear
here soon.

#3 - 04/07/2013 05:33 AM - Anonymous

So with another apology, I will use this space to write down a few more remarks so that I do not forget about them. My line of thinking was as follows: The first step to the systematic solution of this problem would be to generalize zero. It means that the matrix elements would be required to be of a class that has #zero method defined. As for Matrix.zero and Matrix.empty, I think that there should be an option to tell them what this zero is (or tell them the class that has #zero defined). Or perhaps we could play the abstract algebra terminology and call it "additive_identity_element" instead of "zero". That would mean that Matrix would require its elements to comply with at least monoid definition for matrix addition and multiplication, and monoids necessarily need to have the additive identity element defined.

#4 - 04/08/2013 05:53 AM - zzak (Zachary Scott)

Hi Boris, I think it takes too long to read through all of your
messages, so best to define a smaller proposal with exact details for
each feature you want.

On Sat, Apr 6, 2013 at 4:33 PM, boris_stitnicky (Boris Stitnicky)
boris@iis.sinica.edu.tw wrote:

> Issue #8223 has been updated by boris_stitnicky (Boris Stitnicky).
>
> # So with another apology, I will use this space to write down a few more remarks so that I do not forget about them. My line of thinking was as follows: The first step to the systematic solution of this problem would be to generalize zero. It means that the matrix elements would be required to be of a class that has #zero method defined. As for Matrix.zero and Matrix.empty, I think that there should be an option to tell them what this zero is (or tell them the class that has #zero defined). Or perhaps we could play the abstract algebra terminology and call it "additive_identity_element" instead of "zero". That would mean that Matrix would require its elements to comply with at least monoid definition for matrix addition and multiplication, and monoids necessarily need to have the additive identity element defined.
>
> Feature #8223: Make Matrix more omnivorous.
> https://bugs.ruby-lang.org/issues/8223#change-38316
>
> Author: boris_stitnicky (Boris Stitnicky)
> Status: Open
> Priority: Normal
> Assignee:
> Category:
> Target version:
>
> Let's imagine a class Metre, whose instances represent physical magnitudes in metres.
>
> ```
> class Metre
>   attr_reader :magnitude
>   def initialize magnitude; @magnitude = magnitude end
>   def to_s; magnitude.to_s + ".m" end
> end
> ```
>
> Let's say that metres can be multiplied by a number:

```
class Metre
  def * multiplicand
    case multiplicand
    when Numeric then Metre.new( magnitude * multiplicand )
    else
      raise "Metres can only be multiplied by numbers, multiplication by #{multiplicand.class} attempted!"
    end
  end
end
```

And that they can be summed up with other magnitudes in metres, but, as a feature,
not with numbers (apples, pears, seconds, kelvins...).

```
class Metre
  def + summand
    case summand
    when Metre then Metre.new( magnitude + summand.magnitude )
    else
      raise "Metres can only be summed with metres, summation with #{summand.class} attempted!"
    end
  end
end
```

Now with one more convenience constructor Numeric#m:

```
class Numeric
  def m; Metre.new self end
end
```

We can write expressions such as

```
3.m + 5.m
#=> 8.m
3.m * 2
#=> 6.m
```

And with defined #coerce:

```
class Metre
  def coerce other; [ self, other ] end
end
```

Also this expression is valid:

```
2 * 3.m
#=> 6.m
```

Before long, the user will want to make a matrix of magnitudes:

```
require 'matrix'
mx = Matrix.build 2, 2 do 1.m end
#=> Matrix[[1.m, 1.m], [1.m, 1.m]]
```

It works, but the joy does not last long. The user will fail miserably if ze wants to perform matrix multiplication:

```
cv = Matrix.column_vector [1, 1]
mx * cv
#=> RuntimeError: Metres can only be summed with metres, summation with Fixnum attempted!
# where 2.m would be expected
```

In theory, everything should be O.K., since Metre class has both metre summation and multiplication by a number defined. The failure happens due to the internal workings of the Matrix class, which assumes that the elements can be summed together with numeric 0. But it is a feature of metres, that they are picky and allow themselves to be summed only with other Metre instances.

In my real physical units library that I have written, I have solved this problem by
defining an über zero object that produces the expected result, when summed with objects, that would otherwise not lend themselves to summation with ordinary numeric 0,
and patching the Matrix class so that it uses this über zero instead of the ordinary one.

But this is not a very systematic solution. Actually, I think that the Matrix class would be more flexible, if, instead of simply using 0, it asked the elements of the matrix what their zero is, as in:

```
class << Metre
  def zero; new 0 end
end
```

> But of course, that would also require that ordinary numeric classes can tell what their zero is, as in:
>
> ```
> def Integer.zero; 0 end
> def Float.zero; 0.0 end
> def Complex.zero; Complex 0.0, 0.0 end
> # etc.
> ```
>
> I think that this way of doing things (that is, having #zero methods in numeric classes and making Matrix actually require the class of the objects in it to have public class method #zero defined) would make everything more consistent and more algebra-like. I am having this problem for already almost half a year, but I only gathered courage today to encumber you guys with this proposal. Please don't judge me harshly for it. I have actually already seen something like this, in particular with bigdecimal's Jacobian (http://ruby-doc.org/stdlib-2.0/libdoc/bigdecimal/rdoc/Jacobian.html), which requires that the object from which the Jacobian is computed implements methods #zero, #one, #two etc. Sorry again.
>
> --
> http://bugs.ruby-lang.org/

#5 - 04/08/2013 12:21 PM - marcandre (Marc-Andre Lafortune)

- Category set to lib

- Assignee set to marcandre (Marc-Andre Lafortune)

- Priority changed from Normal to 3

Summary: I could consider injecting nothing instead of 0, but can not consider a generic SomeClass.zero. I don't feel your example is a good justification of a need of not injecting 0.

boris_stitnicky (Boris Stitnicky) wrote:

> Let's imagine a class Metre, whose instances represent physical magnitudes in metres.
>
> ```
> class Metre
>   attr_reader :magnitude
>   def initialize magnitude; @magnitude = magnitude end
>   def to_s; magnitude.to_s + ".m" end
> end
> ```
>
> Let's say that metres can be multiplied by a number:
> And that they can be summed up with other magnitudes in metres, but, as a feature,
> not with numbers (apples, pears, seconds, kelvins...).

Shouldn't 0 be specially allowed, i.e 1.m + 0 == 1.m?

> And with defined #coerce:
>
> ```
> class Metre
>   def coerce other; [ self, other ] end
> end
> ```

This is not the way to define coerce. I'm sorry there are not precise specs for coercion, but the idea is to return [transform(other), transform2(self)] such that both elements are compatible. You are reversing the order of things! AFAIK, there is no guarantee this will continue to work as you want it to.

Check how the Matrix class does it (by using an intermediary Scalar class).

But I'm not convinced you have the right approach with the Metre class. Have you tried defining instead a class MeasureWithPhysicalUnit (feel free to shorten the name :-)). You'll need a class "PhysicalUnit" too.

In that case, it would be easy to coerce a numeric value:

```
class Metre
   def coerce other; [ MeasureWithPhysicalUnit.new(other, ''), self ] end
end
```

This probably will be a bit more complex. This should allow a more sensible way of working, and 2.m * 3.m would == 6.m^2

> It works, but the joy does not last long. The user will fail miserably if ze wants to perform matrix multiplication:

Right. Just allow addition with 0, which you have zero reason not to implement.

> But this is not a very systematic solution. Actually, I think that the Matrix class would be more flexible, if, instead of simply using 0, it asked the elements of the matrix what their zero

Definitely not necessary. As you realized in a later email, the 0 is injected in case of empty matrices. If this really was a problem for you, it would be easy to inject the first term in the summation instead (and handle the empty matrix case another way). In no case could a Metre.zero be used, because there is Meter to get in case of zero matrices...

#6 - 04/09/2013 12:23 AM - Anonymous

@Marc-Andre:

> Summary: I could consider injecting nothing instead of 0,
> but can not consider a generic SomeClass.zero. I don't feel
> your example is a good justification of a need of not injecting 0.
> ...
> Shouldn't 0 be specially allowed, i.e 1.m + 0 == 1.m?

I tried that in desperation long ago, but it fails: Due to 0 + 1.m, 0 has to
be treated specially also in #coerce. But 1.m.coerce(0) can't return
[0.m, 1.m], or 0 * 1.m would return 0.m² (zero square metres) instead of
expected 0.m. It is theoretically possible to make coerce return an object
that distinguishes between operators, but that's a lot of work. I'd like to
work on that with Ilya, http://bugs.ruby-lang.org/issues/7604

> ...
> This is not the way to define coerce. I'm sorry there are not precise specs for coercion...

Obviously, simply reversing order will fail upon any noncommutative operation (eg. 0 / 1.m
will fail instead of returning expected 0.m⁻¹). In the real gem, I have it figured, I'll
put it on Github soon. I just wanted to give a simple example here.

> Check how the Matrix class does it (by using an intermediary Scalar class).

I read it once, but it takes some time to get into everything. I definitely plan to read it
more, I consider it one of the crucial libraries which I have to be fluent in.

> But I'm not convinced you have the right approach with the Metre class. Have you tried
> defining instead a class MeasureWithPhysicalUnit (feel free to shorten the name :-)).
> You'll need a class "PhysicalUnit" too.

Again, this was a toy example. I wrote a big fat physical units gem for my simulator, I'll
put it on Github soon. Dimension, Quantity, Magnitude, Unit, mixin for Numerics, dimensional
analysis, everything works, with the distinction that I had to patch Matrix.

> ...
> If this really was a problem for you, it would be easy to inject the first term in the
> summation instead (and handle the empty matrix case another way).

I've been sucking it up for a long time. I didn't want to be a bother. but I gradually gained
an impression, that it might be a concern for others. So could you make the solution you
suggest official, pretty please? Not injecting 0 saves one addition per row for non-empty
matrices, am I right?

> In no case could a Metre.zero be used, because there is Meter to get in case of zero matrices...

One would have to specify of what - zero matrix of what does one want. Imagining:

```
a = Matrix.empty 3, 0, over: Float # new syntax proposal, :over option
b = Matrix.empty 0, 1 # while regular syntax still works
a * b
#=> Matrix[[0.0], [0,0], [0,0]]
```

or

```
a = Matrix.empty 3, 0, over: Metre
a * b
#=> Matrix[[0.m], [0.m], [0.m]]
```

On one hand, I as the user can implement this myself in a subclass, without needing
to bother you the StdLib maintainer. On the other hand, I wanted to publicly discuss
the fact, that for matrix multiplication, matrix need to be defined over an algebraic
ring, and by definition, rings must have an additive identity element and a
multiplicative identity element defined. Multiplicative identity would come into
use in methods such as

```
Matrix.identity( n, options )
  if options[:over] then
    Matrix.scalar( n, options[:over].multiplicative_identity )
  else
    Matrix.scalar( n, 1 )
  end
end

# and then
Matrix.identity 3, over: Metre
Matrix[[1, 0.m, 0.m], [0.m, 1, 0.m], [0.m, 0.m, 1]]
```

I have an itch to try to write a subclass MatrixOverAlgebraicRing (or AlgebraicField?).
If I get that far, I'll ask you what will you think about it.

#7 - 04/09/2013 12:35 AM - marcandre (Marc-Andre Lafortune)

boris_stitnicky (Boris Stitnicky) wrote:

> @Marc-Andre:
>
> > Summary: I could consider injecting nothing instead of 0,
> > but can not consider a generic SomeClass.zero. I don't feel
> > your example is a good justification of a need of not injecting 0.
> > ...
> > Shouldn't 0 be specially allowed, i.e 1.m + 0 == 1.m?
>
> I tried that in desperation long ago, but it fails: Due to 0 + 1.m, 0 has to
> be treated specially also in #coerce. But 1.m.coerce(0) can't return
> [0.m, 1.m], or 0 * 1.m would return 0.m² (zero square metres) instead of
> expected 0.m. It is theoretically possible to make coerce return an object
> that distinguishes between operators, but that's a lot of work. I'd like to
> work on that with Ilya, http://bugs.ruby-lang.org/issues/7604

If you stick with Metre class, like in this example, then coerce 0/1 to a homemade Scalar class (like Matrix does).
If you go for the more generic MagnitudeWithUnit, then coerce it to MagnitudeWithUnit(0, SCALAR) or something.
In both cases, it should work fine.

> So could you make the solution you suggest official

I haven't looked at the code yet, nor have I rejected the issue.

Still, if you want to be serious about your library, 0 + 42.some_unit should work, and this is what you should focus on.

#8 - 04/09/2013 03:42 AM - Anonymous

marcandre (Marc-Andre Lafortune) wrote:

> If you stick with Metre class, like in this example, then coerce 0/1 to a homemade Scalar class (like Matrix does).
> If you go for the more generic MagnitudeWithUnit, then coerce it to MagnitudeWithUnit(0, SCALAR) or something.
> In both cases, it should work fine.

I'll take a look at that Scalar, thanks.

> Still, if you want to be serious about your library, 0 + 42.some_unit should work, and this is what you should focus on.

I want to be serious about my library. I want to make it the best of something like 5 other Ruby unit libraries out there.
I must admit that pragmatically, it saves keystrokes in the interactive mode, if the user is allowed to type

7.m.s^{-1} + 1 #=> 8.m^{-1}

rather than having to tediously type

7.m.s^{-1} + 1.m.s^{-1}

It is a perilous feature, because the user needs to keep in mind...

7.km.h^{-1} + 1 #=> 10.6.km.h^{-1}

...what the standard unit of speed is 1.m.s^{-1}, not 1.km.h^{-1}. I still want
to have this feature, but keep it optional, only when the user explicitly
turns it on. The biggest problem with this feature is the necessary coerce
behavior, which will have to return an advanced object with operator-specific
behavior (#+, #-, #, *#/, #**, #== and whatnot) defined. I'll focus on this
in the near future.

It would seem that with 0, there would be no such problems, because
something + 0 is always something. But unfortunately, this does not
hold well with units that have offsets, such as Celsius degrees. So the
user would still have to keep in mind that 0 in the context of temperatures
means 0.K rather than 0.°C. (Of course, filling matrix with Celsius
temperatures is out of question, because only kelvins support addition,
celsius + kelvins give celsius and celsius + celsius raise QuantityError.)