Ruby - Feature #6284

Add composition for procs

04/12/2012 03:21 PM - pabloh (Pablo Herrero)

Status:	Closed	
Priority:	Normal	
Assignee:	nobu (Nobuyoshi Nakada)	
Target version:		

Description

It would be nice to be able to compose procs like functions in functional programming languages:

```
to_camel = :capitalize.to_proc
add_header = ->val {"Title: " + val}

format_as_title = add_header << to_camel << :strip
```

instead of:

```
format_as_title = lambda {|val| "Title: " + val.strip.capitalize }
```

It's pretty easy to implement in pure ruby:

```
class Proc
  def << block
    proc { |*args| self.call( block.to_proc.call(*args) ) }
  end
end
```

Related issues:

Related to Ruby - Feature #13600: yield_self should be chainable/composable	**Rejected**

Associated revisions

Revision a43e967b8d277fee5cf18329b26bc5c31a3e0363 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Implement Proc#* for Proc composition

- proc.c (proc_compose): Implement Proc#* for Proc composition, enabling composition of Procs and Methods. [Feature #6284]
- test/ruby/test_proc.rb: Add test cases for Proc composition.

From: Paul Mucur mudge@mudge.name

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65911 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision a43e967b8d277fee5cf18329b26bc5c31a3e0363 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Implement Proc#* for Proc composition

- proc.c (proc_compose): Implement Proc#* for Proc composition, enabling composition of Procs and Methods. [Feature #6284]
- test/ruby/test_proc.rb: Add test cases for Proc composition.

From: Paul Mucur mudge@mudge.name

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65911 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision a43e967b - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Implement Proc#* for Proc composition

- proc.c (proc_compose): Implement Proc#* for Proc composition, enabling composition of Procs and Methods. [Feature #6284]

- test/ruby/test_proc.rb: Add test cases for Proc composition.

From: Paul Mucur mudge@mudge.name

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65911 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 3b7b70650c744f8d045328f782fcad360bdd9f46 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Support any callable when composing Procs

- proc.c (proc_compose): support any object with a call method rather than supporting only procs. [Feature #6284]
- proc.c (compose): use the function call on the given object rather than rb_proc_call_with_block in order to support any object.
- test/ruby/test_proc.rb: Add test cases for composing Procs with callable objects.
- test/ruby/test_method.rb: Add test cases for composing Methods with callable objects.

From: Paul Mucur paul@altmetric.com

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65913 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 3b7b70650c744f8d045328f782fcad360bdd9f46 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Support any callable when composing Procs

- proc.c (proc_compose): support any object with a call method rather than supporting only procs. [Feature #6284]
- proc.c (compose): use the function call on the given object rather than rb_proc_call_with_block in order to support any object.
- test/ruby/test_proc.rb: Add test cases for composing Procs with callable objects.
- test/ruby/test_method.rb: Add test cases for composing Methods with callable objects.

From: Paul Mucur paul@altmetric.com

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65913 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 3b7b7065 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

proc.c: Support any callable when composing Procs

- proc.c (proc_compose): support any object with a call method rather than supporting only procs. [Feature #6284]
- proc.c (compose): use the function call on the given object rather than rb_proc_call_with_block in order to support any object.
- test/ruby/test_proc.rb: Add test cases for composing Procs with callable objects.
- test/ruby/test_method.rb: Add test cases for composing Methods with callable objects.

From: Paul Mucur paul@altmetric.com

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65913 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision c71cc2db7f4b179e1a204a0045cea2d80c041874 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

Proc#<< and Proc#>>

[Feature #6284]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65914 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision c71cc2db7f4b179e1a204a0045cea2d80c041874 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

Proc#<< and Proc#>>

[Feature #6284]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65914 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision c71cc2db - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

Proc#<< and Proc#>>

[Feature #6284]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65914 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

History

#1 - 04/12/2012 11:25 PM - trans (Thomas Sawyer)

Or

```
format_as_title = ->{ |val| add_header[to_camel[val.strip]] }
```

#2 - 04/12/2012 11:28 PM - trans (Thomas Sawyer)

Also, I think #+ is better.

#3 - 04/13/2012 02:02 AM - pabloh (Pablo Herrero)

trans (Thomas Sawyer) wrote:

> Also, I think #+ is better.

I saw facets has some similar feature that uses #* instead, maybe because it looks a bit closer to Haskell's composition syntax. Nevertheless, I still like #<< better, it feels your are "connecting" the blocks together.

#4 - 04/13/2012 06:26 AM - alexeymuranov (Alexey Muranov)

I would vote for #*. I think #<< is usually changing the left argument (in place).

#5 - 04/13/2012 06:53 AM - aprescott (Adam Prescott)

See also:
http://web.archive.org/web/20101228224741/http://drmcawesome.com/FunctionCompositionInRuby

#6 - 04/13/2012 07:53 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#7 - 04/13/2012 08:59 AM - pabloh (Pablo Herrero)

aprescott (Adam Prescott) wrote:

> See also: http://web.archive.org/web/20101228224741/http://drmcawesome.com/FunctionCompositionInRuby

Maybe #| could be a possibility. (Without implementing #> or #<).

But I find the article's proposition about the chaining order a bit missleading:

```
transform = add1 | sub3 | negate
```

For me that feels more like "piping" add1 to sub3 to negate, from left to right, not the other way around.

If we choose to take that path I think the following code would be a plausible implementation:

```
class Proc
  def | block
    proc { |*args| block.to_proc.call( self.call(*args) ) }
  end
```

```
end

class Symbol
  def | block
    self.to_proc | block
  end
end
```

#8 - 04/13/2012 02:56 PM - alexeymuranov (Alexey Muranov)

What about #* for composing traditionally (right to left) and #| for piping (left to right)? In mathematics, depending of the area and the subject, both ways are used, and some argue that "piping" is more natural than "precomposing". However, when functions are "piped", the arguments are usually on the left: (arguments)(function1 function2).

Update: i think having the both was a bad idea, it would be redundant.

#9 - 04/15/2012 05:10 PM - trans (Thomas Sawyer)

I agree, #* is appropriate for composition.

#10 - 04/16/2012 11:59 AM - pabloh (Pablo Herrero)

alexeymuranov (Alexey Muranov) wrote:

> Update: i think having the both was a bad idea, it would be redundant.

I was going to say the same thing. Having both #* and #| is redundant and also a bit confusing, since #| doesn't really feel to be the opposite operation of #* at any context. We should choose one or the other but not both.
I still like #| (chaining from left to right) a bit better, but I rather have #* than nothing.

#11 - 10/27/2012 10:44 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Feedback

Positive about adding function composition. But we need method name consensus before adding it?
Is #* OK for everyone?

Matz.

#12 - 11/09/2012 06:56 PM - jballanc (Joshua Ballanco)

Might I humbly suggest #<- :

```
to_camel = :capitalize.to_proc
add_header = ->val {"Title: " + val}

format_as_title = add_header <- to_camel <- :strip
```

Seems to have a nice symmetry with #->

#13 - 11/09/2012 08:10 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I think "<-" reads better but I'm ok with '*' as well.

#14 - 11/09/2012 08:23 PM - rohitarondekar (Rohit Arondekar)

I'm with Joshua, I think #<- reads a lot better.

#15 - 11/10/2012 01:25 AM - alexeymuranov (Alexey Muranov)

I think that the meaning of #<- would not be symmetric with the meaning of #->.

Also, in mathematics, arrows are more like relations than operations. When used to describe functions, usually function arguments go to the arrow's tail, function values to the arrow's head, and function's name, for example, goes on top of the arrow.
(In this sense Ruby's lambda syntax would look to me more natural in the form f = (a,b)->{ a + b } instead of f = ->(a,b){ a + b }.)

The main drawback of #* in my opinion is that is does not specify the direction of composition ((f*g)(x) is f(g(x)) or g(f(x))?), but since in Ruby function arguments are written on the right (f(g(x))), i think it can be assumed that the inner function is on the right and the outer is on the left.

Update : Just for reference, here is how it is done in Haskell : http://www.haskell.org/haskellwiki/Function_composition

#16 - 11/10/2012 02:42 AM - marcandre (Marc-Andre Lafortune)

+1 for #*

The symbol used in mathematics for function composition is a circle (∘); the arrows are for the definitions of functions (like lambdas) only, so #<- or whatever make no sense to me.

Finally, the f ∘ g(x) is defined as f(g(x)), so there is no argument there either.

#17 - 11/10/2012 12:06 PM - duerst (Martin Dürst)

marcandre (Marc-Andre Lafortune) wrote:

> +1 for #*
>
> The symbol used in mathematics for function composition is a circle (∘); the arrows are for the definitions of functions (like lambdas) only, so #<- or whatever make no sense to me.

Very good point.

> Finally, the f ∘ g(x) is defined as f(g(x)), so there is no argument there either.

Not true. Depending on which field of mathematics you look at, either (f ∘ g)(x) is either f(g(x)), or it is g(f(x)). The later is in particular true in work involving relations, see e.g. http://en.wikipedia.org/wiki/Composition_of_relations#Definition.

Speaking from a more programming-related viewpoint, f(g(x)) is what is used e.g. in Haskell, and probably in many other functional languages, and so may be familiar with many programmers.

However, we should take into account that a functional language writes e.g. reverse(sort(array)), so it makes sense to define revsort = reverse * sort (i.e. (f ∘ g)(x) is f(g(x))). But in Ruby, it would be array.sort.reverse, so revsort = sort * reverseve may feel much more natural (i.e. (f ∘ g)(x) is g(f(x))).

#18 - 11/10/2012 01:23 PM - phluid61 (Matthew Kerwin)

I agree that (f ∘ g)(x) is g(f(x)) is more intuitive from a purely programmatic point of view. It is "natural" for the operations to be applied left to right, exactly like method chaining.

--
Matthew Kerwin, B.Sc (CompSci) (Hons)
http://matthew.kerwin.net.au/
ABN: 59-013-727-651

"You'll never find a programming language that frees
you from the burden of clarifying your ideas." - xkcd

#19 - 11/10/2012 06:23 PM - alexeymuranov (Alexey Muranov)

phluid61 (Matthew Kerwin) wrote:

> I agree that (f ∘ g)(x) is g(f(x)) is more intuitive from a purely programmatic point of view. It is "natural" for the operations to be applied left to right, exactly like method chaining.

When functions are applied from left to right, the argument is usually (if not always) on the left. The form (x)(fg)=((x)f)g may look awkward (though i personally used it in a math paper), so i think usually the "exponential" notation is preferred: x^(fg) = (x^f)^g, where x^f corresponds to f(x) in the usual notation.

With method chaining, IMO, the "main argument" of a method is the receiver, and it is on the left. Lambdas and Procs are not chained in the same way as method calls.

Update: I agree that the common syntax for calling functions (f(x) rather then (x)f) should not be an obstacle if Ruby decides to consistently multiply functions putting the inner on the left and the outer on the right. Another syntax for calling functions can be invented in the future, or rubists can learn to live with this inconsistency. For example, Ruby (or Matz) can decide to multiply lambdas with the inner on the left and the outer on the right, and add the following syntax:

```
format_as_title = :strip.to_proc * :capitalize.to_proc * lambda { |val| "Title: " + val }
title = " over here " ^ format_as_title   # instead of `title = format_as_title.call(" over here ")`
```

Update 2012-11-11: I was not clear what i meant by "multiplying from left to right". I meant to say: putting the inner function on the left and the outer on the right. I am correcting this phrase.

#20 - 11/10/2012 09:33 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

In Math multiplication is always associative, even for matrix. I.e: (A*B)*C == A*(B*C). If we use * for ∘ (composition) it resembles multiplication. Function composition is analog to matrix multiplication which are commonly used for transformation compositions as well. In fact, function composition is also associative.

So, when representing h = f ∘ g as h = f * g it makes sense to me (although Math preferring a different symbol for multiplication and composition is a good indication that we should consider this as well for Ruby - more on that later on). But Math representation is procedural, not object oriented. If we try to mix both approaches to fit Ruby philosophy this could lead to great confusion.

Ruby can be also used for procedural programming:

```
sqrt = ->(n){ Math.sqrt n }
# Although I agree that (n)->{} would read more natural to me, just like in CoffeeScript
square_sum = ->(a, b) { a*a + b*b }
hypotenuse = sqrt * square_sum
5 == hypotenuse.call 3, 4 # equivalent to: sqrt.call square_sum.call 3, 4
```

This makes total sense to me using procedural notation. I'm not sure how would someone use this using some OO notation instead...

Now with regards to composition notation, I think a different notation could help those reading some code and trying to understand it. Suppose this method:

```
def bad_name(bad_argument_name, b)
  bad_argument_name * b # or bad_argument_name << b
end
```

You can't know beforehand if bad_argument_name is an array, a number or a proc/lambda. If we read this instead:

```
def bad_name(bad_argument_name, b)
  bad_argument_name <- b
end
```

we would then have a clear indication that bad_argument_name is probably a proc/lambda. I know the same argument could be used to differentiate << between strings and arrays among other cases. But I think that function composition is conceptually much different from those other operations (concatenation, multiplication) than concatenation (<<) is for strings and arrays. In both cases we are concatenating but concatenation means different things for strings and arrays in non surprising ways.

But then using this arrow notation I would expect that (a <- b) would mean "a before b" (b(a(...))) while (a ∘ b) means "a after b" (a(b(...))).

I find it a bit awful to use "hypotenuse = square_sum <- sqrt", although it is the way OO usually work ([4, 5].square_num.sqrt - pseudo-code of course). But we would not be using "[4, 5].hypotenuse", but "hypotenuse.call 4, 5", right? So, since we're using procedural notation for procs/lambdas we should be thinking of procedural programming when deciding which operator to use.

I would really prefer to have lambda syntax as "double = <-{|n| n * 2}" and function composition as "hypotenuse = sqrt -> square_sum" (sqrt after square_sum). But since I don't believe the lambda syntax won't ever change, let's try to see this over a different perspective.

Instead of reading (a <- b) as "a before b", I'll try to think of it as being "b applied to a" (a(b(...))). This also make sense to me so I can easily get used to this. It would work the same way as "*" but there would be a clear indication that this refers to function composition rather than some generic multiplication algorithm.

Having said that, I'd like to confirm that I'm ok with either * or <- and I'd really like to have function composition as part of Ruby.

#21 - 11/24/2012 10:31 AM - mame (Yusuke Endoh)

- Target version changed from 2.0.0 to 2.6

#22 - 12/02/2012 04:53 AM - rits (First Last)

proc composition is not commutative, so the operator should:

1. not imply commutativity
2. not conceal the order of application

i.e. the operator should be visually asymmetrical with clear directionality

e.g. <<, <<<, <-

```
a << b << c = a(b(c(x)))
```

perhaps it also makes sense to have the other direction: c >> b >> a = a(b(c(x)))

#23 - 12/05/2012 12:13 PM - Anonymous

+1 to #*.
+1 to rosenfeld's first 2 paragraphs (h = f ∘ g as h = f * g, and matrix multiplication analogy).
-1 to "<-". Rationale: It is too easy invent a guitar with one more string. Furthermore, when it comes to operators, I consider design by jury a weak approach.

#24 - 12/05/2012 07:47 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I play a 7-string guitar and I can tell you that the extra string greatly improves our possibilities and it is pretty common in Samba and Choro Brazilian music styles:

http://www.youtube.com/watch?v=3mTdpRY6yMI
http://www.youtube.com/watch?v=_FNDXcVr1Pk (here we not only have a 7-string guitar but also a 10-string bandolim while the usual one has 8 strings)

I'm not against #*. I just slightly prefer "<-" over "*".

#25 - 12/05/2012 08:23 PM - alexeymuranov (Alexey Muranov)

rits (First Last) wrote:

> proc composition is not commutative, so the operator should:
>
> 1. not imply commutativity

In algebra, multiplication is rarely commutative, see for example http://en.wikipedia.org/wiki/Quaternion or http://en.wikipedia.org/wiki/Group_(mathematics)

#26 - 06/14/2015 04:55 PM - mudge (Paul Mucur)

- File 0001-proc.c-Implement-Proc-for-Proc-composition.patch added

- File 0002-proc.c-Implement-Method-for-Method-composition.patch added

Attached patches for Proc#* and Method#* for Proc and Method composition including test cases. Also raised as a pull request on GitHub at https://github.com/ruby/ruby/pull/935

One thing that might be worth discussing is the necessity of the type checking: should it be possible to compose any callable object (rather than explicitly Procs and Methods)? The Ruby implementation suggested in the description of this issue suggests calling to_proc on the given argument but we could also demand that the supplied argument responds to call, e.g. in pure Ruby:

```
class Proc
  def *(g)
    proc { |*args, &blk| call(g.call(*args, &blk)) }
  end
end
```

vs.

```
class Proc
  def *(g)
    proc { |*args, &blk| call(g.to_proc.call(*args, &blk)) }
  end
end
```

#27 - 06/23/2015 02:47 PM - mudge (Paul Mucur)

- File 0003-proc.c-Support-any-callable-when-composing-Procs.patch added

Attached patch to support composing with any object that responds to call (rather than raising a TypeError if the object was not a Proc or Method), e.g.

```
class Foo
  def call(x, y)
    x + y
  end
end

f = proc { |x| x * 2 }
g = f * Foo.new

g.call(1, 2) #=> 6
```

#28 - 06/29/2015 08:39 AM - mudge (Paul Mucur)

Regarding the syntax: I also support * as the operator where f * g = f(g(x)) (as it seems close enough to the mathematical syntax already used by other languages such as Haskell and Idris) but if that is too divisive, we could choose a method name from the mathematical definition (https://en.wikipedia.org/wiki/Function_composition) instead:

> The notation g∘f is read as "g circle f ", or "g round f ", or "g composed with f ", "g after f ", "g following f ", or "g of f", or "g on f ".

This opens up the following options:

- Proc#compose: f.compose(g) #=> f(g(x))
- Proc#after: f.after(g) #=> f(g(x))
- Proc#following: f.following(g) #=> f(g(x))
- Proc#of: f.of(g) #=> f(g(x))
- Proc#on: f.on(g) #=> f(g(x))

#29 - 06/29/2015 09:32 PM - pabloh (Pablo Herrero)

It would be nice to be able to compose functions in both ways, like in F#, you can do g << f or g >> f, sadly this was rejected before.

I would settle to have Proc#* for "regular" composition and Proc#| for "piping".
Last time there was no consensus about the syntax. Hopefully we can manage to solve this before 2.3 is released.

#30 - 06/30/2015 08:21 AM - duerst (Martin Dürst)

I'm teaching Haskell in a graduate class, so I'm quite familiar with function composition and use it a lot, but the original example isn't convincing at all. For me, in Ruby, something like val.strip.capitalize reads much, much better than some artificially forced function composition. If there were a method String#prepend, it would be even more natural: val.strip.capitalize.prepend('Title: ').

If there are better examples that feel more natural in Ruby, please post them here.

#31 - 06/30/2015 08:35 AM - Eregon (Benoit Daloze)

Martin Dürst wrote:

> I'm teaching Haskell in a graduate class, so I'm quite familiar with function composition and use it a lot, but the original example isn't convincing at all. For me, in Ruby, something like val.strip.capitalize reads much, much better than some artificially forced function composition. If there were a method String#prepend, it would be even more natural: val.strip.capitalize.prepend('Title: ').
>
> If there are better examples that feel more natural in Ruby, please post them here.

There is String#prepend, but it mutates the receiver.

#32 - 06/30/2015 11:37 AM - pabloh (Pablo Herrero)

Martin Dürst wrote:

> I'm teaching Haskell in a graduate class, so I'm quite familiar with function composition and use it a lot, but the original example isn't convincing at all. For me, in Ruby, something like val.strip.capitalize reads much, much better than some artificially forced function composition. If there were a method String#prepend, it would be even more natural: val.strip.capitalize.prepend('Title: ').
>
> If there are better examples that feel more natural in Ruby, please post them here.

I don't believe you need a pure PF language to benefit from a feature like this. Many ETL projects like transproc (https://github.com/solnic/transproc) would probably find it useful too.

#33 - 06/30/2015 12:39 PM - mudge (Paul Mucur)

Pablo Herrero wrote:

> I don't believe you need a pure PF language to benefit from a feature like this. Many ETL projects like transproc (https://github.com/solnic/transproc) would probably find it useful too.

Transproc is what actually inspired me to submit a patch here: my hope is that having functional composition in the Ruby language itself will enable easier data pipelining using only Procs, Methods and other objects implementing call. The presence of curry (http://ruby-doc.org/core-2.2.2/Proc.html#method-i-curry) seems like a good precedent for adding such functional primitives to the core.

#34 - 07/01/2015 03:52 PM - tomstuart (Tom Stuart)

I support the proposed Proc#*/Method#* syntax and semantics.

The feature being added is **function composition**; not relation composition, not method chaining. Its target audience is most likely to read f * g as “f after g”, so that’s how it should work. Perhaps some Ruby programmers will not use this feature directly (as with Proc#curry) because they neither program nor think in a functional style, but it should be designed to be useful and familiar to those who do. The proposed implementation achieves that.

The asterisk isn’t ideal, but it’s the best choice available.

#35 - 08/07/2015 07:21 AM - systho (Philippe Van Eerdenbrugghe)

Martin Dürst wrote:

> I'm teaching Haskell in a graduate class, so I'm quite familiar with function composition and use it a lot, but the original example isn't convincing at all. For me, in Ruby, something like val.strip.capitalize reads much, much better than some artificially forced function composition. If there were a method String#prepend, it would be even more natural: val.strip.capitalize.prepend('Title: ').
>
> If there are better examples that feel more natural in Ruby, please post them here.

I fully agree with you that a good example really helps to understand the problem which lead to a better solution.

The few times I've missed function composition is ruby is always when I wanted to replace the same pattern :

```
res = step1(base)
res = step2(res)
res = step3(res)
...
res
```

I hate having to mutate the res variable and I would hate even more creating different temp variables especially when I do not know how many steps there will be. I would like to be able to combine the steps then apply the step combination to *base*
There are a lot of examples for this pattern so here is one : data retrieval with ActiveRecord (I assume most people know this library but this is only an example)

```
messages  = all_messages
messages = restrict_to_owner(messages, owner)
filters.each do |filter|
  messages = filter.apply(messages)
end
messages = messages.order(created_at: :asc)
messages = paginate(messages)
```

#36 - 12/30/2015 11:15 AM - mudge (Paul Mucur)

- File deleted (0002-proc.c-Implement-Method-for-Method-composition.patch)

#37 - 12/30/2015 11:15 AM - mudge (Paul Mucur)

- File deleted (0001-proc.c-Implement-Proc-for-Proc-composition.patch)

#38 - 12/30/2015 11:15 AM - mudge (Paul Mucur)

- File deleted (0003-proc.c-Support-any-callable-when-composing-Procs.patch)

#39 - 12/30/2015 11:18 AM - mudge (Paul Mucur)

- File proc-compose.patch added

With the recent addition of Hash#to_proc and performance improvements to Procs in 2.3.0, I have rebased my patch (attached) to add composition between Procs, Methods and other objects responding to call with * in the hope of reviving this proposal.

c.f. https://github.com/ruby/ruby/pull/935

#40 - 12/30/2015 12:28 PM - mudge (Paul Mucur)

- File 0001-proc.c-Implement-Proc-for-Proc-composition.patch added

- File 0002-proc.c-Implement-Method-for-Method-composition.patch added

- File 0003-proc.c-Support-any-callable-when-composing-Procs.patch added

#41 - 12/30/2015 12:28 PM - mudge (Paul Mucur)

- File deleted (proc-compose.patch)

#42 - 10/09/2016 02:30 PM - why-capslock-though (Alexander Moore-Niemi)

I wrote a gem with a C extension of Proc#compose: https://github.com/mooreniemi/proc_compose#usage

What motivated me was map f (map g xs) = map (f . g) xs, and what I still don't understand (being a newbie to extending Ruby or understanding its internals) is that .map(&some_proc).map(&some_other_proc) still behaves better than .map(&(some_proc * some_other_proc)) given my current implementation of compose. Although I think composition has a lot of uses, the admittedly small but free performance benefit I expected to gain was top of my list.

I do think emphasis on composition suggests a somewhat different style of writing Ruby, but I think it can be a good one, actually.

Paul: what's the performance of your compose? If I have time later I can use https://github.com/mooreniemi/graph-function to try and see.

#43 - 10/11/2016 08:18 PM - mudge (Paul Mucur)

Alexander Moore-Niemi wrote:

> Paul: what's the performance of your compose? If I have time later I can use https://github.com/mooreniemi/graph-function to try and see.

I ran the following benchmark with benchmark-ips:

```
require 'benchmark/ips'

double = proc { |x| x * 2 }
quadruple = proc { |x| x * 4 }
octuple = double * quadruple
inline_octuple = proc { |x| x * 2 * 4 }
nested_octuple = proc { |x| quadruple.call(double.call(x)) }

numbers = [1, 2, 3, 4, 5]

Benchmark.ips do |x|
  x.report('composing procs') do
    numbers.map(&octuple)
  end

  x.report('chaining procs') do
    numbers.map(&double).map(&quadruple)
  end

  x.report('single proc') do
    numbers.map(&inline_octuple)
  end

  x.report('nested proc') do
    numbers.map(&nested_octuple)
  end

  x.compare!
end
```

And also see a performance drop with composition over chaining multiple maps:

```
Warming up --------------------------------------
     composing procs    27.822k i/100ms
      chaining procs    32.096k i/100ms
         single proc    49.021k i/100ms
         nested proc    27.337k i/100ms
Calculating -------------------------------------
     composing procs    341.874k (± 0.5%) i/s -      1.725M in   5.045764s
      chaining procs    389.031k (± 0.7%) i/s -      1.958M in   5.032912s
         single proc    666.544k (± 0.6%) i/s -      3.333M in   5.001266s
         nested proc    321.919k (± 0.8%) i/s -      1.613M in   5.010562s

Comparison:
         single proc:   666543.8 i/s
      chaining procs:   389031.4 i/s - 1.71x  slower
     composing procs:   341873.8 i/s - 1.95x  slower
         nested proc:   321919.1 i/s - 2.07x  slower
```

It might be interesting to look at object allocations as we effectively create a nested Proc which might account for the slow down.

#44 - 10/12/2016 08:28 AM - mudge (Paul Mucur)

Yukihiro Matsumoto wrote:

> Positive about adding function composition. But we need method name consensus before adding it?
> Is #* OK for everyone?

Aside from implementation details, is the lack of consensus on the method name (and the resulting behaviour re left vs right operand) the main blocker here?

#45 - 10/12/2016 09:46 AM - duerst (Martin Dürst)

Paul Mucur wrote:

> Yukihiro Matsumoto wrote:
>
> > Positive about adding function composition. But we need method name consensus before adding it?
> > Is #* OK for everyone?
>
> Aside from implementation details, is the lack of consensus on the method name (and the resulting behaviour re left vs right operand) the main blocker here?

If Matz says so, then yes, this is the main blocker.

#46 - 01/22/2017 06:50 PM - thorstenhirsch (Thorsten Hirsch)

fyi: Composing procs is way faster now. It beats all other algorithms in Paul's benchmark on ruby 2.3.3:

```
Warming up --------------------------------------
     composing procs   135.483k i/100ms
      chaining procs    55.595k i/100ms
         single proc    84.448k i/100ms
         nested proc    48.095k i/100ms
Calculating -------------------------------------
     composing procs      2.363M (± 6.2%) i/s -     11.787M in   5.011859s
      chaining procs    701.936k (± 3.4%) i/s -      3.558M in   5.074972s
         single proc      1.207M (± 3.6%) i/s -      6.080M in   5.044891s
         nested proc    579.005k (± 2.9%) i/s -      2.934M in   5.071310s

Comparison:
     composing procs:  2362658.0 i/s
         single proc:  1206768.2 i/s - 1.96x  slower
      chaining procs:   701936.2 i/s - 3.37x  slower
         nested proc:   579005.4 i/s - 4.08x  slower
```

Tested on linux and macOS (same result).

#47 - 03/13/2017 08:13 AM - matz (Yukihiro Matsumoto)

I want to make sure if everyone agrees with "*" instead of OP's "<<".
Besides that I also wanted to mention that Koichi concerns that function composition may be far slower than method chaining.

Matz.

#48 - 04/28/2017 07:47 PM - RichOrElse (Ritchie Buitre)

matz (Yukihiro Matsumoto) wrote:

> I want to make sure if everyone agrees with "*" instead of OP's "<<".
> Besides that I also wanted to mention that Koichi concerns that function composition may be far slower than method chaining.
>
> Matz.

+1 for #*

Initially I thought of the F# convention #<< and it's counter part #>> as intuitive. But after giving it some thought, and practice, I prefer #* and #+.

```
class Proc
  def +(other) # forward compose operation
    other.to_proc * self
  end

  def *(other) # backward compose operation
    -> arg { self.call other.(arg) } # with only one argument for performance reason.
  end
end

f = -> n { n  * 2 }
g = -> n { n  * 4 }
h = f * g                           # equivalent to (g + f)

puts h.(5)                          #=> 40
puts  (h  + :odd?).call(3)          #=> false
```

Instead of composition I see potential use for shovel operations #<< and #>> for piping purposes similar to Elixir's #|>.

```
class Object
  def >>(transform) # piping operation
    transform.(self)
  end
end

class Integer
  def >>(num_or_func)
    return self << -num_or_func unless num_or_func.respond_to? :call
    super
  end
end

class Proc
  def <<(input) # feed value
    call input
  end
end

add_header = -> val {"Title: #{val}" }
format_as_title = add_header + :capitalize + :strip

puts 'Title goes here.' >> format_as_title  #=> Title: title goes here.
puts format_as_title << 'Title goes there.' #=> Title: title goes there.
puts 100 >> format_as_title                 #=> Title: 100
puts 100 >> 2                               #=> 25
```

#49 - 05/27/2017 01:35 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #13600: yield_self should be chainable/composable added

#50 - 08/31/2017 06:26 AM - yuroyoro (TOMOHITO Ozaki)

Most languages do not define function composition in built-in operators, but provide them as function or method such as compose.

F.Y.I) https://rosettacode.org/wiki/Function_composition

In some few languages defined function composition operator as following.

haskell : .
F# : << and >>
Groovy : << and >>

I think Ruby should proivide compose method (and and_then as forward composition), and alias some operator (like #<< or `#*) to them.

In my opinion, +1 for << and >> instead of *.
Because there is no language define function composition as * and +, but F# and groovy defined as << and >>. It is intutive.

By the way, It is useful if Symbol#<< (or Symbol#+) is shortcut method to sym.to_proc.compose(other_proc).

```
arr.map(&:upcase.to_proc.compose(:to_s.to_proc))
arr.map(&:to_s >> :upcase)
```

It is more visually and intuitive.

My reference implemenation of composition is following.

https://github.com/yuroyoro/ruby/pull/7

#51 - 01/29/2018 07:49 AM - zverok (Victor Shepelev)

Please take this to next Developers Meeting Agenda?

The best possible name is discussed for 6 years already, and feature seems pretty nice to have. I believe that sometimes it is just good to make "executive decision" about name once and for all, at least better than postpone feature for years.

(I still unhappy with yield_self name, though :))

#52 - 04/20/2018 06:34 PM - baweaver (Brandon Weaver)

yuroyoro (TOMOHITO Ozaki) wrote:

> Most languages do not define function composition in built-in operators, but provide them as function or method such as compose.

F.Y.I) https://rosettacode.org/wiki/Function_composition

In some few languages defined function composition operator as following.

haskell : .
F# : << and >>
Groovy : << and >>

I think Ruby should proivide compose method (and and_then as forward composition), and alias some operator (like #<< or `#*) to them.

In my opinion, +1 for << and >> instead of *.
Because there is no language define function composition as * and +, but F# and groovy defined as << and >>. It is intutive.

By the way, It is useful if Symbol#<< (or Symbol#+) is shortcut method to sym.to_proc.compose(other_proc).

```
arr.map(&:upcase.to_proc.compose(:to_s.to_proc))
arr.map(&:to_s >> :upcase)
```

It is more visually and intuitive.

My reference implemenation of composition is following.

https://github.com/yuroyoro/ruby/pull/7

I do like the idea of using shovel as it's already fairly common in the language, and present in others. I'd be interested in potentially making a container variant though:

```
arr.map(&:to_s >> :upcase)

# becomes
arr.map(&[:to_s, :upcase])
```

Though that would involve introducing the idea of Array#to_proc. I've used a similar technique in a few gems with #[](), so a vague implementation may look like:

```
class Array
  def to_proc
    self[1..].reduce(self[0].to_proc, :compose)
  end
end
```

Wherein it'd be nice if #compose tried to coerce its argument:

```
def compose(sym)
  fn = sym.is_a?(Proc) ? sym : sym.to_proc

  ...
end
```

Currently planning on releasing a gem this weekend that will do a similar thing, it could serve as inspiration for an implementation, but my C is no where near good enough to try at the Ruby core level.

Aside: Have we ever considered implicitly to_procing anything passed to a method expecting a block? e.g. arr.map(&:to_s) becomes arr.map(:to_s)

#53 - 04/30/2018 06:51 PM - baweaver (Brandon Weaver)

I just realized that infix operators evaluate before to_proc, allowing something like this:

```
[1,2,3].map(&:succ.to_proc >> :to_str.to_proc)
```

Codified it into a gem for kicks: https://github.com/baweaver/mf

#54 - 05/17/2018 05:41 AM - nobu (Nobuyoshi Nakada)

- Description updated

#55 - 05/17/2018 05:56 AM - matz (Yukihiro Matsumoto)

- Status changed from Feedback to Open

Considering the combination of OOP and FP, it seems a good idea to adding both forward and reverse combination of procs. So we pick Groovy way (adding << and >> methods to Proc).

We need more discussion if we would add combination methods to the Symbol class.

Matz.

#56 - 07/30/2018 08:13 AM - printercu (Max Melentiev)

matz (Yukihiro Matsumoto) wrote:

> Considering the combination of OOP and FP, it seems a good idea to adding both forward and reverse combination of procs. So we pick Groovy way (adding << and >> methods to Proc).

What do you think about selecting only single operation? Here is Groovy example from provided link:

```
final times2 = { it * 2 }
final plus1 = { it + 1 }

final plus1_then_times2 = times2 << plus1
final times2_then_plus1 = times2 >> plus1
```

It looks like << is less intuitive and readable. I'm also really afraid of having chance to read something like a >> b << c >> d.

#57 - 08/08/2018 05:41 AM - ko1 (Koichi Sasada)

- Assignee changed from matz (Yukihiro Matsumoto) to nobu (Nobuyoshi Nakada)

#58 - 08/08/2018 06:37 PM - shan (Shannon Skipper)

matz (Yukihiro Matsumoto) wrote:

> We need more discussion if we would add combination methods to the Symbol class.
>
> Matz.

A little sugar on Symbol does look really nice:
https://gist.github.com/havenwood/d305b42f5b542e9de1eaa8e56ba6bdd7#file-compose_procs-rb-L32-L45

#59 - 08/10/2018 07:58 PM - shan (Shannon Skipper)

I used this proposal along with #14781 for a fun solution to a problem proposed in the #ruby irc channel:
https://gist.github.com/havenwood/b6d55b412fbf583758c91b8ee339a822

I like these features a lot and love that you can just implement them in the meanwhile. <3 Ruby!

#60 - 11/12/2018 12:43 PM - nobu (Nobuyoshi Nakada)

I've forgotten to post the patch to use << and >>.
https://github.com/nobu/ruby/tree/feature/6284-proc-composition

#61 - 11/14/2018 05:33 PM - pabloh (Pablo Herrero)

nobu (Nobuyoshi Nakada) wrote:

> I've forgotten to post the patch to use << and >>.
> https://github.com/nobu/ruby/tree/feature/6284-proc-composition

Is adding composition methods to the Symbol class still being considered?

#62 - 11/15/2018 02:22 AM - nobu (Nobuyoshi Nakada)

pabloh (Pablo Herrero) wrote:

> Is adding composition methods to the Symbol class still being considered?

No, I don't think that we consider a Symbol object a method.

#63 - 11/22/2018 05:51 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset trunk|r65911.

---

proc.c: Implement Proc#* for Proc composition

- proc.c (proc_compose): Implement Proc#* for Proc composition, enabling composition of Procs and Methods. [Feature #6284]
- test/ruby/test_proc.rb: Add test cases for Proc composition.

From: Paul Mucur mudge@mudge.name

Files

0001-proc.c-Implement-Proc-for-Proc-composition.patch	3.65 KB	12/30/2015	mudge (Paul Mucur)
0002-proc.c-Implement-Method-for-Method-composition.patch	2.67 KB	12/30/2015	mudge (Paul Mucur)
0003-proc.c-Support-any-callable-when-composing-Procs.patch	3.97 KB	12/30/2015	mudge (Paul Mucur)