Ruby trunk - Feature #12854

Proc#curry should return an instance of the class, not Proc

10/19/2016 11:44 PM - zenspider (Ryan Davis)

Status:	Feedback	
Priority:	Normal	
Assignee:		
Target version:		

Description

```
class ChainedProc < Proc
end

ChainedProc.new { |x, y, z| 42 }.curry.class # => Proc
```

History

#1 - 10/20/2016 02:02 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Feedback

What's the rationale or the use-case?
Curried proc is not a subset of the original proc, I think.

#2 - 10/20/2016 02:13 AM - nobu (Nobuyoshi Nakada)

- Tracker changed from Bug to Feature

#3 - 10/20/2016 05:44 AM - zenspider (Ryan Davis)

```
class ChainedProc < Proc
  # ... this stuff here is the use-case
end
```

#4 - 10/20/2016 05:45 AM - zenspider (Ryan Davis)

Another (albeit, poor/strange) example:

https://gist.github.com/tenderlove/ca673b3ce18460890cfd18e09bb1657c

#5 - 12/21/2016 02:37 PM - shyouhei (Shyouhei Urabe)

We briefly looked at this issue at today's developer meeting.

Practically, it is possible to change what #curry returns. But attendees were not sure if such change skew the conceptual nature of currying operation.

#6 - 12/21/2016 04:04 PM - mame (Yusuke Endoh)

I proposed and first implemented Proc#curry. I have no idea how this change may skew the conceptual nature of the currying operation. Do you have any particular concern?

#7 - 12/22/2016 01:17 AM - shyouhei (Shyouhei Urabe)

Yusuke Endoh wrote:

> I proposed and first implemented Proc#curry. I have no idea how this change may skew the conceptual nature of the currying operation. Do you have any particular concern?

No particular problem was shown at the meeting, as far as I remember. Other attendees might have such thing though.

#8 - 01/10/2017 02:06 AM - crb002 (Chad Brewbaker)

Defining complex functions with curry would be nontrivial:

```
double = ->(a)(a+a)
g = ->(a,b,c){ f.call(1,c, double.call(a))}
```

I suggest adding Proc#trans and Proc#lens.

Proc#trans applies a transformation (repeated permutation) to the argument list.

Proc#lens is list of functions applied to the argument list. Think of a constant as a function that takes zero arguments.

```
class Proc
   def trans(*args)
       ->(*a){
           a.flatten!
           bound = [a.size, args.size].min
           alist = (0...bound).collect{|i| a[args[i] ] }
           self.call(alist)
       }
   end

   def lens(*args)
       concretes = [Integer,TrueClass,FalseClass,String,Float,Symbol,Array,Hash]
       ->(*a){
           a.flatten!
           bound = [a.size, args.size].min
           alist = (0...bound).collect{|i|
               if(concretes.include?(args[i].class))
                  args[i]
                else
                   args[i].call(a[i])
               end
           }
           self.call(alist)
       }
   end
end

id = ->(*args){args.inspect}

f = id.trans(0,1,2)
p f.call(0,1,2) == "[[0, 1, 2]]"

g = id.trans(2,0,0)
p g.call() == "[[]]"
p g.call(0)  == "[[nil]]"
p g.call(0,1) == "[[nil, 0]]"
p g.call(0,1,2) == "[[2, 0, 0]]"
p g.call(0,1,2,3,4) == "[[2, 0, 0]]"

single = ->(a){a}
double = ->(a){a+a}
triple = ->(a){a+a+a}
q = id.lens(single, double,triple)
p q.call(5,5,5) == "[[5, 10, 15]]"
p q.call(1,2,3) == "[[1, 4, 9]]"

h = q.lens(single,double,triple)
p h.call(5,5,5) == "[[5, 20, 45]]"
k = id.lens(single, 444,triple)
p k.call(4,4,4,4,4) == "[[4, 444, 12]]"
```

Link, pull reqs welcome, https://github.com/chadbrewbaker/endoscope/blob/master/catgist.rb

#9 - 01/10/2017 01:29 PM - crb002 (Chad Brewbaker)

Another pattern is Proc#multilens(tin, lenses, tout); where "tin" is a transformation from the input argument list to lenses, "lenses" are the intermediate functions, and "tout"is a mapping from the intermediate functions to an output argument list.

```
id = ->(*a){*a}

add = -> (a,b}{a+b}
decrement = -> (a){a-1}
q = id.multilens([0,0,1],[add,decrement],[1,0])
q(1,2,3) == [add.call(1,1),decrement.call(2)].trans(1,0) == [2,1].trans(1,0) == [1,2]
```

There is also a conveyor belt pattern, but I haven't thought of a good syntax. There are k FIFO queues. The lenses are stacked across the queues in a directed acyclic graph. You also have buffer lenses that can allow inputs to keep flowing for a fixed amount. Yes, I played a lot of Factorio over Christmas :) https://wiki.factorio.com/Belt_transport_system

If you can draw a belt pattern on a doughnut with d holes without crossing; that says something about the complexity of how many cores you can run it in parallel on, and how hard it is to map on an FPGA. Hoping to add SIMD, OpenCL, and FPGA support to Fiddle so lambdas can run at full

machine bandwidth on basic types.

Also, for more complexity you could drop the DAG requirement and have arbitrary directed graphs; allowing the convayor belts to be recurrent. Also, buffers could be arbitrary Ruby data structures allowing for even more flexibility. Conveyor belts happen in the real world where you have a speed of light latency connection. The number of packets that can be in flight is the distance divided by the latency of placing a packet on the wire. At the speed of light two processors placed in the opposite corners of your laptop (30cm apart) can only ping-pong around 500 million times a second even if their latency to process a packet is Plank's constant.

#10 - 01/19/2017 05:54 AM - matz (Yukihiro Matsumoto)

Chad, is this issue what you really want? Or you want new methods like #trans and #lens?
If you still want to make #curry return the subclass, I expect a use case.
If you want #trans and #lens, submit a new issue.

Matz.

#11 - 10/25/2017 09:59 PM - dsisnero (Dominic Sisneros)

don't use lens as the name because this is a common used name in functional programming for a different concept