Proc#with and Proc#by, for partial function application and currying

Proc#by allows currying implicitly

class Proc
def by(*head)
  return self if head.none?
  curry(head.size.next).(*head)
end
class Method
def by(*head)
  to_proc.by(*head)
end
class Symbol
def by(*head)
  to_proc.by(*head)
end
double = :*.by(2) # => proc { |n| 2 * n }

Proc#with pre-defines trailing arguments and/or block.

class Proc
def with(*tail, &blk)
  if arity == tail.size.next
    proc { |head| call head, *tail, &blk }
  else
    proc { |*head| call *head, *tail, &blk }
  end
end
class Method
def with(*head, &blk)
  to_proc.with(*head, &blk)
end
class Symbol
def with(*head, &blk)
  to_proc.with(*head, &blk)
end
double = :*.with(2) # => proc { |n| n * 2 }

That's the basic idea, but I've also expanded on it by optimising and defining operators (+, &, |) and other methods (Proc#such) here.
I am not sure if the API seems ok. I am also not sure if matz wants to have Symbols have methods such as .with(). For example, to me personally it is not entirely clear why "with 2" would be equal to "n * 2" as such.

I am also not sure about the use case - it has not been mentioned in this issue as far as I can see.

However had, perhaps we should wait a bit on the upcoming developer meeting this year anyway, because there have been other proposed changes that are somewhat related to the issue of how much class Symbol should be able to do - e. g. see what Victor Shepelev suggested, linked in to https://bugs.ruby-lang.org/issues/15229 for Symbol#call.

Then we also know matz' opinion about class Symbol in regards to any possible changes to it.

shevegen (Robert A. Heiler) wrote:

I am not sure if the API seems ok. I am also not sure if matz wants to have Symbols have methods such as .with(). For example, to me personally it is not entirely clear why "with 2" would be equal to "n * 2" as such.

Thank you for taking the time to review my proposal and for the suggestions.

To illustrate more clearly how Symbol#with works, here's another example:

```
[DateTime.new(2018,11,1), DateTime.new(2018,11,30)].map(&:strftime.with("%m/%d/%Y"))
```

Which is the same as the following:

```
[DateTime.new(2018,11,1), DateTime.new(2018,11,30)].map { |d| d.strftime("%m/%d/%Y") }
```

Although #15229Symbol#call is shorter than the functional equivalent proposal Symbol#with, the later's interface is consistent with Proc#with and Method#with where, as you are already aware, have their method call already taken.

shevegen (Robert A. Heiler) wrote:

I am also not sure about the use case - it has not been mentioned in this issue as far as I can see.

Here's a use case for filling optional arguments.

Given a method named greet:

```
def greet(name, greeting = 'hello')
  p "#{greeting.capitalize}! #{name}"
end
greet 'bob' # => "Hello! bob"
```

We can reuse the same method by pre-filling the last argument like so:

```
module Spanish
  GREETINGS = method(:greet).with('hola')
  # Using Method#call would invoke the method instead of returning a Proc.
end

Spanish::GREETINGS['Roberto'] # => "Hola! Roberto"
```

Matz.

matz (Yukihiro Matsumoto) wrote:

This kind of partial evaluation is an interesting idea, but as a non-native speaker, I wonder those words do not cause confusion which works which way? At least I was confused.

Matz.
I wonder those words do not cause confusion which works which way? At least I was confused.

I agree with your assessment Matz. Both 'with' and 'by' are such flexible words, they're the first words that came to my mind. Unfortunately they are also too flexible, making them vague.

**Descriptive Names**

Until the community decides on more useful aliases, for now picking descriptive method names such as 'partial' and 'targets' is less confusing.

```ruby
class Proc
  def partial(*tail, &blk)
    proc { |head| call(*head, *tail, &blk) }
  end

  def targets(*head)
    curry(head.size.next)[*head]
  end
end
```

**Alternative Prepositions**

Even though I am partial to (pun intended) the 'with' interface, I ruminated on finding alternative words. So far I've stumbled upon these prepositions which looks promising.

**Proc#in** for implicit currying.

```ruby
multiply = -> x, y { x * y }
double = multiply.in(2) # => proc { |n| multiply.(2, n) }
```

**Proc#on** for partial evaluation.

```ruby
divide = -> x, y { x / y }
half = divide.on(2) # => proc { |n| divide.(n, 2) }
```

I like the pairing of 'in' with 'on'. Aside from being only 2 characters long, they allow to mentally map the arguments placement.

General information are placed first using 'in'.

```ruby
to_s = :to_s.proc          # => proc { |x, *options| x.to_s(*options) }
ten_to_base = to_s.in(10)  # => proc { |base| to_s.(10, base) }
five_to_base = to_s.in(5)  # => proc { |base| to_s.(5, base) }
```

Specific or optional details are placed last using 'on'.

```ruby
to_binary = to_s.on(2)     # => proc { |n| to_s.(n, 2) }
to_hexadecimal = to_s.on(16)  # => proc { |n| to_s.(n, 16) }
```

On the downside, in some context, the word 'on' is less natural sounding to an English speaker compared to the more flexible word 'with'.

```ruby
method(:greet).on("Kon'nichiwa")
```

On the upside the word 'in' is less redundant and won't be confused with the '_by' idioms.

```ruby
find_person_by = :find_by.in(Person) # => proc { |*criteria| Person.find_by(*criteria) }
```