Ruby - Feature #16264

Real "callable instance method" object.

10/19/2019 09:06 AM - zverok (Victor Shepelev)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

It is a part of thinking about the "argument-less call style" I already made several proposals about.

Preface

Argument-less call style is what I now call things like map(&:foo) and each(&Notifications.:send) approaches, and I believe that naming the concept (even if my initial name is clamsy) will help to think about it. After using it a lot on a large production codebase (not only symbols, but method references too, which seem to be less widespread technique), I have a strong opinion that it not just "helps to save the keypresses" (which is less important), but also helps to clearer separate the concepts on a micro-level of the code. E.g. if you feel that each(&Notifications.:send) is "more right" than select { |el| Notifications.send(el, something, something) }, it makes you think about Notifications.send design in a way that allows to pass there *exactly* that combination of arguments so it would be easily callable that way, clarifying modules responsibilities.

> (And I believe that "nameless block parameters", while helping to shorter the code, lack this important characteristic of clarification.)

The problem

One of the problems of "argument-less calling" is passing additional arguments, things like those aren't easy to shorten:

```
ary1.zip(ary2, ary3).map { |lines| lines.join("\n") }
#                                    ^^^^
construct_url.then(&HTTP.:get).body.then { |text| JSON.parse(text, symbolize_names: true) }
#                                                 ^^^^^^^^^^^^^^^^^^^^^^
```

(BTW, here is a blog post where I show recently found technique for solving this, pretty nice and always existing in Ruby, if slightly esotheric.)

There's a lot of proposals for "partial applications" which would be more expressive than .curry (guilty myself), but the problematic part in all of this proposals is:

> **The most widespread "shortening" is &:symbol, and Symbol itself is NOT a functional object, and it is wrong to extend it with functional abilities.**

One of consequences of the above is, for example, that you can't use 2.6's proc combination with symbols, like File.:read >> :strip >> :reverse. You want, but you can't.

Here (while discussing aforementioned blog posts), I stumbled upon an idea of how to solve this dilemma.

The proposal

I propose to have a syntax for creating a functional object that when being called, sends the specified method to its first argument. Basically, what Symbol#to_proc does, but without "hack" of "we allow our symbols to be convertible to functional objects". Proposed syntax:

```
[1, 2, 3].map(&.:to_s)
```

Justification of the syntax:

- It is like Foo.:method (producing functional object that calls method)
- Orphan .:method isn't allowed currently (you need to say self.:method to refer to "current selfs method"), and Matz's justification was "it would be too confusable with :method, small typo will change the result" -- which in PROPOSED case is not as bad, as

:foo and .:foo both meaning the same thing;
- It looks kinda nice, similar to (proposed and rejected) map { .to_s } → with my proposal, it is map(&.:to_s), implying somehow applying .to_s to the previous values in the chain.

The behavior: .:foo produces object of class, say, MethodOfArgument (class name is subject to discuss) — which makes differences of "Proc created from Symbol" (existing internally, but almost invisible) obvious and hackable.

Potential gains

- New object could be used in proc composition: File.:read >> .:strip >> JSON.:parse >> .:compact

- When both "method" and "method of argument" are proper functional objects, a new partial application syntax can be discussed, common for them both. For example (but **not necessary this method name!**)

paragraph_hashes.map(&.:merge.with(author: current_author))
filenames.map(&File.:read.with(mode: 'rb'))

```
* (I believe at this point we'll be able to finally switch from discussing "show we extend Symbol with more callable-alike functionality" to just method's name and exact behavior)
* I am not an expert, but probably some optimizations could be applied, too
* Currently, `:sym.to_proc` is internally different from other proc, but this can't be introspected:

```ruby
:read.to_proc.inspect # => "#<Proc:0x0000556216192198(&:read)>"
                                                  # ^^^^^
```

- Probably, exposure of this fact could lead to some new interesting metaprogrammin/optimization techniques.

Transition

:foo and .:foo could work similarly for some upcoming versions (or indefinitely), with .:foo being more powerful alternative, allowing features like groups_of_lines.map(&.:join.partial_apply(' ')) or something.

It would be like "real" and "imitated" keyword arguments. "Last hash without braces" was good at the beginning of the language lifecycle, but then it turned out that real ones provide a lot of benefits. Same thing here: &:symbol is super-nice, but, honestly, it is semantically questionable, so may be slow switch to a "real thing" would be gainful for everybody?..

History

#1 - 10/19/2019 01:54 PM - nobu (Nobuyoshi Nakada)

- Description updated

- Status changed from Open to Feedback

zverok (Victor Shepelev) wrote:

> **Transition:** :foo and .:foo could work similarly for some upcoming versions (or indefinitely), with .:foo being more powerful alternative, allowing features like groups_of_lines.map(&.:join.with(' ')) or something.

Is the point .:join.with(' ')?

#2 - 10/19/2019 02:09 PM - zverok (Victor Shepelev)

@nobu (Nobuyoshi Nakada), sorry, I am not sure I get your question.

Are you asking about how .:join.with(' ') is supposed to work?
It is (theoretical and maybe not the best) example of how "partial application" could look in Ruby. It is not part of my proposal, just an idea of "**if** we'll have proper functional object .:join, we can discuss how idiomatic partial application could work on *functional object* (blocks, methods, and method-of-arg proposed)".

#3 - 10/20/2019 12:00 PM - shevegen (Robert A. Heiler)

I hope it is ok to link to one of your blog entries (for those who may ask - I am not affiliated with zverok nor did I ask him about it prior to that; it may just provide some more context to the larger topic at hand):
```

https://zverok.github.io/blog/2019-10-18-each_with_object.html

To the topic itself here. Hmmmmmm. Personally I am a bit biased against it, largely due to syntax considerations. I actually think that &.:method is not very pretty.

To be fair: I think that .map(&:strip) is not very pretty either, but it's sort of like an "established standard" since many years in ruby. I also use it myself sometimes, e. g. especially when I do strip/chop/chomp. I am also aware of people trying to find a way to use succinct syntax while also being able to pass in arguments, such as the example the example given:

```
&.:join.with(' ')
```

This may be a lot up to one's individual style and preferences, but I don't quite like it. I don't think that many are against the functionality, e. g. any succinct representation may be useful, but there is some trade-off in regards to syntax and semantics.

> similar to (proposed and rejected) map { .to_s }

The syntax of e. g. .map { .method } was quite clean, cleaner than most other suggestions. :) But I did not like the implication in semantics ... we have the object.method notation in general, and suddenly we could omit the leading object part. That would be quite strange in my opinion.

I think what nobu is specifically asking is about the proposal for the specific syntax that would go with e. g. the part where you would pass additional arguments to the method(s) involved.

> as :foo and .:foo both meaning the same thing

I am biased because I am much preferring :foo over .:foo if only due to bad eyesight alone :) - but that is also a reason why I dislike &.: even more.

Perhaps I am just too lazy; on the other hand, I would prefer ruby and ruby's syntax to not become too complicated. With &.: we now would have three characters. How many more would we want? Five? Eight? Why stop at eight? :P

It is not solely about syntax and semantics alone, though; some of it is about simplicity versus complexity. I understand that this is up to one's preference a lot. I just tend to prefer simplicity usually because my brain is not easily able to master complexity.

By the way, I agree with this statement:

> Symbol itself is NOT a functional object, and it is wrong to extend it with functional abilities.

I think it is best to retain Symbol as close as possible to matz' original concept for Symbol, even though over time Symbol was a bit extended. It is understandable that there are general suggestions of people wanting to extend Symbol, but I think it is better to retain Symbol to be a simpler concept than to extend it into many ways as a "dynamic" but complex concept.

> And I believe that "nameless block parameters", while helping to shorter the code, lack this important characteristic of clarification.

I think the use cases are a bit different, although it also depends on the definition(s). I was not aware of an older suggestion, or the comment by Jeremy, so the use cases may differ (or "elegance of syntax"). For quick debugging I think not having to rely on the names is useful; I do however had also agree with benoit to some extent in the sense that some ruby users may happily use them all over the place, and it's not so pretty or useful then, IMO. I still think it is different to the proposal here, though - but if we would reason about syntax alone then oldschool ruby syntax beats all of these suggestions from a "pretty" point of view, including "nameless" param style. ;) (Although it was also said in the past that while an elegant, succinct syntax is great to have, it is not necessarily ruby's primary design point alone. I just like clean and simple syntax whenever it is possible.)

Martin pointed out that some of the "functional" ideas are a bit disparate, so a larger concept that could cover the individual suggestions may be better. I don't have a useful suggestion for that, though, partially also because the applications are a bit dissimilar to how I may tend to use ruby (more from an "oldschool" OOP use mostly).

#4 - 10/22/2019 02:24 AM - Dan0042 (Daniel DeLorme)

It sounds like an interesting way to improve functional programming in ruby. I can definitely see the point of having a proper functional syntax instead of trying to hack everything via symbols. But I'm not sure it's the best way. At the core, I believe this was best explained by duerst:

> we should stop dealing with individual ideas of how to improve functional programming in Ruby and work out an overview of what's missing and how to address it, understanding that Ruby is first and foremost an OO language and we'll never get to the same point as Haskell or something similar.

So I think that having a shorthand .:foo syntax as equivalent of :foo.to_proc can definitely be part of the solution, but ideally it would be considered together with other elements of a functional grand master plan.

> similar to (proposed and rejected) map { .to_s }

> The syntax of e. g. .map { .method } was quite clean, cleaner than most other suggestions.

That makes my heart ache all over again T_T

#5 - 10/23/2019 08:09 AM - zverok (Victor Shepelev)

- Description updated

Updated description significantly to clarify intentions.

#6 - 10/23/2019 02:02 PM - osyo (manga osyo)

Is .:hoge a :hoge.to_proc syntax sugar?
And, is map(&self.:hoge) and map(&.:hoge) different in meaning?
In Ruby, self.hoge and hoge without self often have the same meaning
But, it looks very strange that map(&self.:hoge) and map(&.:hoge)` have different meanings.

#7 - 10/23/2019 03:53 PM - zverok (Victor Shepelev)

> is .:hoge a :hoge.to_proc syntax sugar?

No, and that's a core of the proposal. :hoge.to_proc produces regular Proc (which is different internally, but conceals this fact). .:hoge is meant to produce new core object.

> In Ruby, self.hoge and hoge without self often have the same meaning
> But, it looks very strange that map(&self.:hoge) and map(&.:hoge) have different meanings.

Yes, that's an obvious compromise for this proposal. The intuition of "you could drop self." is already broken for self.:hoge, which is prohibited currently; I believe it would be a "gotcha", yet bearable one.

#8 - 12/20/2019 07:15 AM - matz (Yukihiro Matsumoto)

- Status changed from Feedback to Closed

We have suspended the idea of .: operator, so this proposal is no longer meaningful. We may revisit the method reference operator in the future. We have to remember this PR as well. FYI, we have numbered parameters now, so the proposed [1, 2, 3].map(&.:to_s) can be expressed by [1, 2, 3].map{_1.to_s}. This could lower the value of this proposal.

Matz.

#9 - 12/27/2019 08:51 PM - Dan0042 (Daniel DeLorme)

As a note for the future:

Since unary operations like -@ and ~@ use the @ sigil to represent the receiver, I think it would make sense for the callable instance method object to have a syntax like @.hoge