Ruby master - Feature #16113

Partial application

08/19/2019 10:07 AM - zverok (Victor Shepelev)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Preface: One of the main "microstructures" of the code we use is chaining methods-with-blocks; and we really love to keep those blocks DRY when they are simple. Currently, for DRY-ing up simple blocks, we have:

- foo(&:symbol)
- foo(&some.method(:name)) (as of 2.7, foo(&some.:name))
- Currently disputed "nameless block args": foo { something(@1) } or foo { something(@) } or foo { something(it) }

Proposal: I argue that short and easy-to-remember partial application of blocks and methods can make methods-with-blocks much more pleasant and consistent to write, and continue softly shifting Ruby towards "functional" (while staying true to language's spirit).

In order to achieve this, I propose method {Symbol,Method,Proc}#w (from with), which will produce Proc with last arguments bound.

Example of usability:

```
# No-shortcuts: fetch something and parse as JSON:
fetch(urls).map { |body| JSON.parse(body) }
# Could be already (2.7+) shortened to:
fetch(urls).map(&JSON.:parse)

# But if you have this:
fetch(urls).map { |body| JSON.parse(body, symbolize_names: true) }
# How to shorten it, to don't repeat body?
# "Nameless block args" answer:
fetch(urls).map { JSON.parse(@1, symbolize_names: true) }
# Partial application answer:
fetch(urls).map(&JSON.:parse.w(symbolize_names: true))
```

I believe that the latter (while can be easily met with usual "hard to understand for a complete novice") provides the added value of producing proper "functional object", that can be stored in variables and constants, and generally lead to new approaches to writing Ruby code.

Another example:

```
(6..11).map(&:**.w(2)).map(&:clamp.w(20, 50))
# => [36, 49, 50, 50, 50, 50]
```

Reference implementation:

```
class Symbol
  def w(*args)
    proc { |receiver, *rest| receiver.send(self, *rest, *args) }
  end
end

class Method
  def w(*args)
    proc { |receiver, *rest| self.call(receiver, *rest, *args) }
  end
end

class Proc
  def w(*args)
    prc = self
    proc { |*rest| prc.call(*rest, *args) }
```

```
  end
end
```

History

#1 - 08/19/2019 11:11 AM - shevegen (Robert A. Heiler)

Personally I dislike this proposal primarily due to the name alone. I don't think a method named .m is good in this context.

We do have short named methods here and there, of course, such as "p" or "pp", but I feel that these cases are simpler, and relate mostly to "output-related tasks".

To me it is not clear why a method called "m" should signify "partial application".

There are a few other aspects intermingled in the proposal, or comments that I think are a bit strange.

For example:

```
foo(&some.method(:name)) (as of 2.7, foo(&some.:name))
Currently disputed "nameless block args": foo { something(@1) } or foo { something(@) } or foo { something(it)
 }
```

First, I don't think the second is "disputed", but let's ignore this for the moment. To me, foo(&some.method(:name)) has little to do with e. g. foo { method(@1) }. Both syntax-wise and from the functionality; but syntax-wise, I actually think that ALL the syntax examples are not hugely elegant. Neither is the oldschool foo(&:symbol). I use the latter myself, mostly because it leads to shorter code, but I think it is visually not as elegant as the longer block variant:

```
.each {|foo|
}
```

In my opinion, this style is the clearest. Not as short as & but clearer and cleaner, in my opinion. But I digress.

Another part of the proposal mentioned the "staying true to language's spirit". I think this is a bit problematic, though. Ruby was always multi-paradigm and had influences from different languages. My own opinion is that ruby's biggest strength is the focus on OOP, but I think this may depend a lot on how someone may use ruby. My use cases will be different from what other people use. There is also a lot of ruby code out in the wild I would never write, use or want to maintain. :)

Some of these coding styles are, IMO, somewhat orthogonal or opposing to one another if applied at the same time. But this is admittedly also a LOT due to the individual preference of the ruby user at hand.

Another aspect that I dislike is that suggestions like this, but also similar ones, increase the complexity of ruby by a LOT.

Consider this:

```
fetch(urls).map(&JSON.:parse.w(symbolize_names: true))
```

Honestly, I don't even want to have to get my brain to want to decipher this. Again, this is up to a personal opinion and preference, just as people may appreciate a shorter syntax to block parameters or dislike it - but if possible, I think it would be better to strive for simplicity rather than build up more and more complexity into ruby in general.

This is of course only my own personal opinion on that matter.

Note that using a slightly different name other than e. g. w, may partially alleviate the above problem, but the syntax complexity and overall complexity may still remain.

IMO I think this is simply for matz to decide how much complexity ruby should strive to in general.

This is also, as stated, up to an individual's preference - zverok

loves deeply nested, chained blocks. I prefer them to be much much simpler, including syntax. Difficult to unite two somewhat opposing views on the same topic. :D

#2 - 08/19/2019 12:23 PM - Hanmac (Hans Mackowiak)

the Devs should maybe look at #curry for this, currently it doesn't support a way to curry keyword arguments

#3 - 08/20/2019 01:28 AM - shan (Shannon Skipper)

An aside, but I took a stab at a pure Ruby implementation of keyword argument currying:
https://gist.github.com/havenwood/db041566abeac894602c188c77374040

```
['{"aim":true}', '{"impossible":false}'].map &JSON.:parse.curry.(symbolize_names: true)
#=> [{:aim=>true}, {:impossible=>false}]
```

#4 - 08/23/2019 11:51 PM - shan (Shannon Skipper)

I like Hanmac's idea.

```
Klass.:meth.curry.()
```

Seems very close to:

```
Klass.:meth.w()
```

I know it has previously been said that #curry is a bit of an easter egg. Would it be acceptable to add keyword argument support for #curry?

zverok (Victor Shepelev) If #curry gets support for keyword arguments, would it suffice for this case or do you think #w would still be called for?

#5 - 08/24/2019 10:57 AM - zverok (Victor Shepelev)

> If #curry gets support for keyword arguments, would it suffice for this case

I don't think so, unfortunately.

1. Of my examples, it covers only JSON.parse
   - It doesn't cover non-keyword arguments, which, I believe, it should. Example: construct_filename.then(&File.:read.w('rb'))
   - It doesn't cover symbol partial application. While it *could* be seen as "esotheric" by some, but in realistic code, there are pretty regular demand for something like numbers.map(&:+.w(2)). I once proposed the numbers.map(&:+.(2)) syntax, which is "nice-looking", but semantically incorrect
2. While "conceptually true to functional programming", it is just *too long*, which kinda undermines the whole point.