

Ruby - Feature #16166

Remove exceptional treatment of *foo when it is the sole block parameter

09/13/2019 08:17 AM - sawa (Tsuyoshi Sawada)

Status:Closed Priority:Normal Assignee: Target version:	
Description In the parameter signature of a code block for a method that is not involved in method definition or creation of lambda objects, two types of arguments ["a"] and "a" are neutralized: <pre>instance_exec(["a"]){ foo, bar foo} # => "a" instance_exec("a"){ foo, bar foo} # => "a"</pre> <pre>instance_exec(["a"]){ *foo, **bar foo} # => ["a"] instance_exec("a"){ *foo, **bar foo} # => ["a"]</pre> This is the same behavior as with assignment constructions: <pre>foo, bar = ["a"]; foo # => "a" foo, bar = "a"; foo # => "a"</pre> <pre>*foo = ["a"]; foo # => ["a"] *foo = "a"; foo # => ["a"]</pre> And it contrasts with constructions involved in method definition or creation of lambda objects, where the distinction is preserved: <pre>lambda{ foo foo}.call(["a"]) # => ["a"] lambda{ foo foo}.call("a") # => "a"</pre> <pre>->(foo){foo}.call(["a"]) # => ["a"] ->(foo){foo}.call("a") # => "a"</pre> <pre>lambda{ *foo foo}.call(["a"]) # => ["a"] lambda{ *foo foo}.call("a") # => ["a"]</pre> <pre>->(*foo){foo}.call(["a"]) # => ["a"] ->(*foo){foo}.call("a") # => ["a"]</pre> However, when *foo is the sole parameter of a code block for a method that is not involved in method definition or creation of lambda objects, ["a"] and "a" are not neutralized: <pre>instance_exec(["a"]){ *foo foo} # => ["a"] instance_exec("a"){ *foo foo} # => ["a"]</pre> behaving in contrast to assignment constructions, and rather on a par with constructions involved in method definition or creation of lambda objects. Particularly, existence or absence of another parameter **bar entirely changes what foo refers to: <pre>instance_exec(["a"]){ *foo foo} # => ["a"] instance_exec(["a"]){ *foo, **bar foo} # => ["a"]</pre> I find this behavior inconsistent and confusing. I would like to request to remove this exceptional treatment of splatted parameter *foo when it is the sole parameter in a code block. I request this behavior: <pre>instance_exec(["a"]){ *foo foo} # => ["a"]</pre>	

History

#1 - 09/13/2019 08:18 AM - sawa (Tsuyoshi Sawada)

- Description updated

#2 - 09/13/2019 08:21 AM - sawa (Tsuyoshi Sawada)

- Subject changed from Remove exceptional handling of **foo* when it is the sole block parameter to Remove exceptional treatment of **foo* when it is the sole block parameter

- Description updated

#3 - 09/13/2019 10:16 AM - mame (Yusuke Endoh)

I agree that Ruby's arguments are insanely complex. In the basic case, "a" and ["a"] are distinguished.

```
p instance_exec("a") { |foo| foo } #=> "a"
p instance_exec(["a"]){ |foo| foo } #=> ["a"]

p instance_exec("a") { |*foo| foo } #=> ["a"]
p instance_exec(["a"]){ |*foo| foo } #=> ["a"]
```

In *some* cases, they are not distinguished.

```
p instance_exec("a") { |foo, bar| foo } #=> "a"
p instance_exec(["a"]){ |foo, bar| foo } #=> "a"

p instance_exec(["a"]){ |*foo, **bar| foo } #=> ["a"]
p instance_exec("a") { |*foo, **bar| foo } #=> ["a"]
```

The rule is fairly complex or even inconsistent. I cannot understand [the condition](#).

I have no opinion which case `|*foo|` should belong to. (I personally hope that `|*foo, **bar|` belongs to the same case as `|*foo|` because keywords are separated from positional arguments.)

Anyway, I don't think that it is a good idea to change the behavior just because it is inconsistent. We need an evidence that the behavior actually confuses many people, at least.

#4 - 09/13/2019 03:44 PM - shevegen (Robert A. Heiler)

We need an evidence that the behavior actually confuses many people, at least.

It does not confuse me because ... I try to avoid it altogether. :D

I think sawa's issue can be a bit shortened (sorry!) to the last comparison:

```
instance_exec(["a"]){ |*foo| foo } # => ["a"]
instance_exec(["a"]){ |*foo| foo } # => ["a"]
```

Although I may miss (or not completely understand) all of the reasoning, I think that change would make sense (to me) - but I may not understand the consequences.

I only remember even matz having fun in a presentation with the whole keyword arg situation before. ;) (One reason why I try to actually avoid keywords is because I find them more difficult to deal/cope with than oldschool options hash. But I guess this may differ from ruby user to ruby user since it is a personal preference.)

Perhaps there should be a simple and consistent rule for how `*` and `**` is to be interpreted at all times, including what should happen if both are used at the same time. What I find indeed a bit confusing is that `*` changes if `**` is also used. That part is very strange to me personally. Might also be mentioned in the documentation, but for me personally, I gladly stick to the simpler variants. :D

#5 - 09/13/2019 03:45 PM - shevegen (Robert A. Heiler)

Actually that reminds me - mame mentioned that the ruby core team needs a motivation/impetus if a change is necessary based on real usage. So I think this may be a good call for ruby users to comment in particular when it may affect them (either way) in actual code. Me personally I am not affected in either way, but it may be a good idea to get a survey/query to ruby users out there to comment, in particular when it may affect them in their own

code base or a code base they use/depend on.

#6 - 09/25/2019 03:00 AM - matz (Yukihiro Matsumoto)

I think the following code behavior is wrong:

```
p instance_exec(["a"]){|*foo, **bar| foo } #=> ["a"]
```

It should return `[["a"]]`.

Matz.

#7 - 09/27/2019 11:03 PM - jeremyevans0 (Jeremy Evans)

matz (Yukihiro Matsumoto) wrote:

I think the following code behavior is wrong:

```
p instance_exec(["a"]){|*foo, **bar| foo } #=> ["a"]
```

It should return `[["a"]]`.

Here's a pull request for that: <https://github.com/ruby/ruby/pull/2502>

Note that it breaks some tests/specs. I believe the reason methods with keywords were handled differently is because the last element of the argument could be used as keywords:

```
$ ruby -e "p proc{|*foo, **bar| [foo, bar]}.call([1, {a: 1}])"
-e:1: warning: The last argument is used as the keyword parameter
-e:1: warning: for `call' defined here
[[1], {a=>1}]
```

As you can see, this now raises a warning in Ruby 2.7, and behavior will change in Ruby 3.0. Do we want to make this change in 2.7, or do we want to wait for 3.0?

#8 - 10/22/2019 06:00 PM - Dan0042 (Daniel DeLorme)

I think this is related:

```
proc{ |a,b| [a,b] }.call(1,2)    #=> [1, 2]
proc{ |*ab| ab }.call(1,2)      #=> [1, 2]
proc{ |a,b| [a,b] }.call([1,2]) #=> [1, 2]
proc{ |*ab| ab }.call([1,2])    #=> [[1, 2]]
```

I *really* think the last result should be `[1, 2]`. Otherwise I totally fail to understand the logic. But of course there's always backward compatibility to worry about...

#9 - 11/27/2019 05:55 PM - Eregon (Benoit Daloze)

@sawa wrote this in the dev-meeting ticket:

Unintended arity. This must be fixed in an earlier stage before Ruby 3.

I think matz conclusion is all behavior shown in this bug so far is intended, except for `*foo, **bar`.

@Dan0042

```
proc{ |a,b| [a,b] }.call(1,2)    #=> [1, 2]
proc{ |*ab| ab }.call(1,2)      #=> [1, 2]
proc{ |a,b| [a,b] }.call([1,2]) #=> [1, 2]
proc{ |*ab| ab }.call([1,2])    #=> [[1, 2]]
```

That's just how Proc works, multiple parameters will splat an Array if a single Array argument is given.

A single parameter will not splat. The `*rest` parameter will not splat either, or it would delegate arguments incorrectly.

I think long-term we might want to use lambda semantics by default for blocks, which doesn't have that splatting magic.

#10 - 12/02/2019 03:55 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote:

The `*rest` parameter will not splat either, or it would delegate arguments incorrectly.

Thanks! Finally I can see some meaning behind the madness. Normally I would expect `proc { |*a| }.call([1,2])` to behave like the assignment `*a = [1,2]` but it does not. But now I can see that `proc { |*a| foo(*a) }.call(arg1)` would not work if `arg1` happened to be an array.

However I still agree with sawa and believe that the current behavior for `proc { |*a| }` is confusing. If you want that particular case of delegation to work you could simply use a lambda instead of a proc. Having that special exception where a proc behaves like a lambda just for `|*rest|...` it makes things overly complicated. I mean, even mame says he cannot understand the condition! With all due respect to Matz, this would be much simpler if we could say that all procs behave with assignment semantics, and lambdas with parameter semantics.

Of course the backward compatibility is an issue but that can be handled with proper deprecation warnings. I think that would be a worthwhile change for ruby. My 2¢.

#11 - 01/05/2020 04:06 PM - sawa (Tsuyoshi Sawada)

[The log for developers' meeting 20191128](#) says that Matz hoped this to be done in 2.7. However, Ruby 2.7 still returns this:

```
instance_exec(["a"]){|*foo, **bar| foo } #=> ["a"]
```

Likely forgotten? Perhaps, it can be done by the next release (Not sure about the version name; 2.8 or 3.0).

#12 - 03/09/2020 12:55 AM - jeremyevans0 (Jeremy Evans)

I've rebased the pull request (<https://github.com/ruby/ruby/pull/2502>) on current master. Assuming it passes CI, I plan to commit it, as matz has already indicated his support for the change, and keyword argument separation means that there is no point in autosplatting anymore, since the last value in the array will no longer be treated as keywords.

#13 - 03/09/2020 03:50 AM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Closed

#14 - 03/14/2022 12:41 PM - Eregon (Benoit Daloze)

What's the reason for this behavior?

It seems inconsistent with the resolution of this issue, and it seems nobody would want that behavior:

```
irb(main):005:0> proc { |a| a }.call([1, 2])
=> [1, 2]
irb(main):006:0> proc { |a, **kw| a }.call([1, 2])
=> 1
irb(main):007:0> proc { |a, kw: 42| a }.call([1, 2])
=> 1
```

Also:

```
irb(main):010:0> proc { |a, **kw| a }.call([1, 2])
=> 1
irb(main):011:0> proc { |a, **kw| a }.call([1, 2], **{})
=> [1, 2]
```

Which is really as it shows a difference between passing `**{}` or nothing (a bug I guess?).

#15 - 03/14/2022 05:36 PM - Eregon (Benoit Daloze)

^ filed as a proper issue: <https://bugs.ruby-lang.org/issues/18633>