Ruby trunk - Bug #15620

Block argument usage affects lambda semantic

02/24/2019 06:57 PM - alanwu (Alan Wu)

Status:	Open		
Priority:	Normal		
Assignee:	matz (Yukihiro Matsumoto)		
Target version:			
ruby -v:		**Backport:**	2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

Description

The following snippet demonstrate the issue:

```
def pass_after_use(&block)
  raise unless block
  lambda(&block).call
end

def direct_pass(&block)
  lambda(&block).call
end

pass_after_use do |_arg|
  puts "fine, because block is materialized into a Proc before it is passed to #lambda"
end

direct_pass do |_arg|
  puts "Raises because all args are required. This is not printed"
end
```

Output:

```
fine, because block is materialized into a Proc before it is passed to #lambda
Traceback (most recent call last):
    2: from lambda-block-pass.rb:14:in `<main>'
    1: from lambda-block-pass.rb:7:in `direct_pass'
lambda-block-pass.rb:14:in `block in <main>': wrong number of arguments (given 0, expected 1) (Arg
umentError)
```

I think having the line raise unless block affect Kenrel#lambda's semantic is pretty surprising. Note that if I do raise unless block_given?, call to the lambda without arg also raises.

If I was to decide, I would always have the resulting lambda have required arguments even after multiple levels of block pass. That is, as long as the original block is a literal block.

This is either a breaking change or a regression from 2.4. The same script executes without raising in 2.4.5 (block arguments are always materialized).

History

#1 - 03/05/2019 04:47 AM - marcandre (Marc-Andre Lafortune)

- Assignee set to matz (Yukihiro Matsumoto)

Current behavior is clearly a bug due to the block passing implementation.

It's actually not clear to me why lambda(&proc{}).lambda? returns false. It doesn't seem useful and is counter-intuitive to me.

If others think we could revisit this, it has been clearly documented as such for over 10 years (r14713), so changing this behavior would be a breaking change. I would guess with very little impact. A quick search in the top 500 gems revealed a single use of lambda(&...) which doesn't look incompatible: https://github.com/CocoaPods/CocoaPods/blob/master/lib/cocoapods/resolver.rb#L435

I found two other uses in specs, rubocop: spec/rubocop/cop/style/stabby_lambda_parentheses_spec.rb and in vcr: spec/lib/vcr/structs_spec.rb. Neither seem problematic.

We could deprecate this use with warning and then change it?

Otherwise we could simply fix the regression.

#2 - 03/07/2019 02:57 PM - Eregon (Benoit Daloze)

lambda(&proc{}).lambda? returns false because I think the rule is:
once a Proc is created, it never changes its lambda-ness.

So the only way to create a lambda is passing a block directly to lambda (or through send), or using ->.

So I think direct_pass above should create a non-lambda Proc, and I would argue it's a bug it creates a lambda since MRI 2.5.

#3 - 03/07/2019 04:27 PM - marcandre (Marc-Andre Lafortune)

Eregon (Benoit Daloze) wrote:

> lambda(&proc{}).lambda? returns false because I think the rule is:
> once a Proc is created, it never changes its lambda-ness.

Right. I was wondering why this is the "rule", what's the rationale. It makes lambda(&...) equivalent to ...to_proc but less clear. I was thinking only in terms of parameter passing (in which case the rule feels counter productive), but there's also handling of break and return (in which case the rule makes more sense I guess, still not convinced).

Note that the rule can be broken, for example with:

```
def transform_proc_into_lambda(&proc)
  o = Object.new
  o.singleton_class.define_method(:foo, &proc)
  o.method(:foo).to_proc
end

transform_proc_into_lambda{}.lambda? # => true
```

#4 - 03/07/2019 06:05 PM - Eregon (Benoit Daloze)

Right, define_method is kind of an exception.
However, it doesn't mutate the lambda-ness of the Proc, it creates a new Proc from that block with lambda semantics.

It's still confusing for the user though, as a given literal block could be used as both Proc and lambda, which is likely unexpected (e.g. break/return cannot work correctly in that block).
This can also be achieved with send(:proc or :lambda) { ... }, but that case is more explicit about it.

#5 - 03/07/2019 10:40 PM - alanwu (Alan Wu)

Since normal Ruby methods can't differentiate between a literal block and a block pass, having #lambda behave like a normal method gives us more consistency.
#lambda doesn't need to mutate its argument, it could return a lambda proc based on the block-passed proc.

#6 - 03/11/2019 05:16 AM - ko1 (Koichi Sasada)

we will fix it.

#7 - 03/19/2019 07:03 AM - Hanmac (Hans Mackowiak)

nobu (Nobuyoshi Nakada) should this one be closed too?