Ruby master - Bug #16504

`foo(*args, &args.pop)` should pass all elements of args

01/12/2020 12:47 AM - mame (Yusuke Endoh)

Status:	Closed		
Priority:	Normal		
Assignee:	jeremyevans0 (Jeremy Evans)		
Target version:			
ruby -v:	ruby 2.7.0p0 (2019-12-25 revision 647ee6f091) [x86_64-linux]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN

Description

https://bugs.ruby-lang.org/issues/16500?next_issue_id=16499&prev_issue_id=16501#note-7

```
def foo(*args)
  p args
end

# in 2.7
args = [1, 2, -> {}]; foo(   *args, &args.pop) #=> passes [1, 2] (bug; [1, 2, ->{}] is expected)
args = [1, 2, -> {}]; foo(0, *args, &args.pop) #=> passes [0, 1, 2, ->{}] (good)
```

Related issues:

Related to Ruby master - Bug #16500: Argument is added to both splat and last...	**Closed**
Related to Ruby master - Bug #12860: Splatting an argument does not obey left...	**Closed**

Associated revisions

Revision aae8223c - 06/18/2020 03:19 PM - jeremyevans (Jeremy Evans)

Dup splat array in certain cases where there is a block argument

This makes:

```
  args = [1, 2, -> {}]; foo(*args, &args.pop)
```

call foo with 1, 2, and the lambda, in addition to passing the
lambda as a block. This is different from the previous behavior,
which passed the lambda as a block but not as a regular argument,
which goes against the expected left-to-right evaluation order.

This is how Ruby already compiled arguments if using leading
arguments, trailing arguments, or keywords in the same call.

This works by disabling the optimization that skipped duplicating
the array during the splat (splatarray instruction argument
switches from false to true). In the above example, the splat
call duplicates the array. I've tested and cases where a
local variable or symbol are used do not duplicate the array,
so I don't expect this to decrease the performance of most Ruby
programs. However, programs such as:

```
  foo(*args, &bar)
```

could see a decrease in performance, if bar is a method call
and not a local variable.

This is not a perfect solution, there are ways to get around
this:

```
  args = Struct.new(:a).new([:x, :y])
  def args.to_a; a; end
  def args.to_proc; a.pop; ->{}; end
  foo(*args, &args)
  # calls foo with 1 argument (:x)
  # not 2 arguments (:x and :y)
```

A perfect solution would require completely disabling the

optimization.

Fixes [Bug #16504]
Fixes [Bug #16500]

History

#1 - 01/12/2020 12:52 AM - mame (Yusuke Endoh)

https://github.com/ruby/ruby/pull/2833

#2 - 01/12/2020 12:52 AM - mame (Yusuke Endoh)

- Related to Bug #16500: Argument is added to both splat and last &block argument added

#3 - 01/12/2020 04:37 PM - Eregon (Benoit Daloze)

What's the definition of foo here?

I believe the previous behavior is the block expression gets evaluated before the rest of the arguments.
We should decide and clarify if the block or positional arguments are evaluated first.

If I see foo(*args, &args.pop) I would expect no duplicated arguments.
I believe that's the previous and more intuitive behavior.
I'd be happy if we can warn it though, because that code is really unreadable.

foo(*args, &args.last) should be used if wanting to pass the last argument both as block and last positional.

#4 - 01/12/2020 11:44 PM - mame (Yusuke Endoh)

Okay, I'll ask matz which is right. But I believe that the 2.6 and current behavior is wrong because Ruby has a principle of left-to-right evaluation.
Actually, foo(*ary, ary.pop) was changed between 2.1 and 2.2; 2.2 and later duplicate the last argument. And what do you think about foo(*ary, 42, &ary.pop)?

#5 - 01/13/2020 12:00 AM - mame (Yusuke Endoh)

- Related to Bug #12860: Splatting an argument does not obey left-to-right execution order added

#6 - 01/13/2020 12:01 AM - mame (Yusuke Endoh)

I spent an hour to find the ticket: https://bugs.ruby-lang.org/issues/12860

#7 - 01/13/2020 01:21 PM - Eregon (Benoit Daloze)

Agreed left-to-right would be far more consistent.

We just need to be aware that whoever writes foo(*args, &args.pop) probably expects no duplication (so they would disagree on "bug" probably).
But such code deserves to be clearer IMHO.

#8 - 01/13/2020 02:27 PM - marcandre (Marc-Andre Lafortune)

mame (Yusuke Endoh) wrote:

> Okay, I'll ask matz which is right. But I believe that the 2.6 and current behavior is wrong because Ruby has a principle of left-to-right evaluation.

For sure current behavior is wrong and left-to-right evaluation must be observed.

#9 - 05/28/2020 10:04 PM - jeremyevans0 (Jeremy Evans)

I've added a pull request that fixes this issue: https://github.com/ruby/ruby/pull/3157

It's not a perfect fix, but a perfect fix would require disabling a commonly used optimization, resulting in additional array allocations in common cases when using splatting with block passing.

#10 - 06/14/2020 11:57 AM - Eregon (Benoit Daloze)

Do we have an idea of how much overhead would it be to disable this not-correct optimization?
Arrays are copy-on-write so it seems not so expensive to have an extra array, and also guarantees the callee cannot mutate the original array.
I'd rather we don't compromise semantics because optimizations are too limited.

For the &:symbol case, we could check if Symbol#to_proc is redefined.
For the &local_var case, we could check if it's already a Proc.

#11 - 06/16/2020 04:48 AM - mame (Yusuke Endoh)

- Description updated

#12 - 06/16/2020 05:09 AM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans) wrote in #note-9:

> I've added a pull request that fixes this issue: https://github.com/ruby/ruby/pull/3157

Wow https://bugs.ruby-lang.org/issues/16504#note-1

#13 - 06/16/2020 02:56 PM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote in #note-12:

> jeremyevans0 (Jeremy Evans) wrote in #note-9:
>
> > I've added a pull request that fixes this issue: https://github.com/ruby/ruby/pull/3157
>
> Wow https://bugs.ruby-lang.org/issues/16504#note-1

Sorry, I completely missed that you also had a fix for this (basically the same other than the test added).

#14 - 06/18/2020 05:36 AM - matz (Yukihiro Matsumoto)

Accepted. We can ignore incompatibility here.

Matz.

#15 - 06/18/2020 05:38 AM - mame (Yusuke Endoh)

- Assignee set to jeremyevans0 (Jeremy Evans)

jeremyevans0 (Jeremy Evans) wrote in #note-13:

> mame (Yusuke Endoh) wrote in #note-12:
>
> > jeremyevans0 (Jeremy Evans) wrote in #note-9:
> >
> > > I've added a pull request that fixes this issue: https://github.com/ruby/ruby/pull/3157
> >
> > Wow https://bugs.ruby-lang.org/issues/16504#note-1
>
> Sorry, I completely missed that you also had a fix for this (basically the same other than the test added).

Don't worry. It is good to know that two persons reached the same solution independently. Your patch is better because it includes tests, so could you commit your patch?

#16 - 06/18/2020 03:19 PM - jeremyevans (Jeremy Evans)

- Status changed from Open to Closed

Applied in changeset git|aae8223c7076483f1f16411810887790b2f3a66dd.

Dup splat array in certain cases where there is a block argument

This makes:

```
  args = [1, 2, -> {}]; foo(*args, &args.pop)
```

call foo with 1, 2, and the lambda, in addition to passing the
lambda as a block. This is different from the previous behavior,
which passed the lambda as a block but not as a regular argument,
which goes against the expected left-to-right evaluation order.

This is how Ruby already compiled arguments if using leading
arguments, trailing arguments, or keywords in the same call.

This works by disabling the optimization that skipped duplicating the array during the splat (splatarray instruction argument switches from false to true). In the above example, the splat call duplicates the array. I've tested and cases where a local variable or symbol are used do not duplicate the array, so I don't expect this to decrease the performance of most Ruby programs. However, programs such as:

```
foo(*args, &bar)
```

could see a decrease in performance, if bar is a method call and not a local variable.

This is not a perfect solution, there are ways to get around this:

```
args = Struct.new(:a).new([:x, :y])
def args.to_a; a; end
def args.to_proc; a.pop; ->{}; end
foo(*args, &args)
# calls foo with 1 argument (:x)
# not 2 arguments (:x and :y)
```

A perfect solution would require completely disabling the optimization.

Fixes [Bug #16504]
Fixes [Bug #16500]