

Ruby master - Feature #16253

Shorthand "forward everything" syntax

10/15/2019 03:38 PM - Dan0042 (Daniel DeLorme)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		
Description		
What about using this:		
<pre>def foo(*) @bar.foo(*)</pre>		
to mean this:		
<pre>def foo(*a, **o, &b) @bar.foo(*a, **o, &b)</pre>		
I used def foo(*) because that's currently valid ruby code, but I'm fine with any syntax.		
It's like the no-parentheses super shorthand, but for any method.		
It makes it easier to write correct forwarding code.		
If rubyists must be told they have to change their forwarding code in 2.7 (due to keyword arguments), the pill might be easier to swallow if the change is a reduction rather than an increase in verbosity.		
And we'd even be future-proof if an eventual FOURTH kind of parameter is introduced!!!!		
Related issues:		
Related to Ruby master - Misc #16157: What is the correct and *portable* way ...		Open
Related to Ruby master - Feature #16296: Alternative behavior for `...` in me...		Open
Related to Ruby master - Feature #16378: Support leading arguments together w...		Closed

Associated revisions

Revision 62d43828 - 10/21/2019 05:35 PM - nobu (Nobuyoshi Nakada)

Arguments forwarding [Feature #16253]

Revision 6279e45c - 10/24/2019 03:15 PM - nobu (Nobuyoshi Nakada)

Arguments forwarding is not allowed in lambda [Feature #16253]

Revision d1ae2bc2 - 11/12/2019 08:05 AM - mame (Yusuke Endoh)

NEWS: Make it clear that delegation syntax (...) requires parentheses

Ref [Feature #16253]

History

#1 - 10/15/2019 03:51 PM - Eregon (Benoit Daloze)

- Related to Misc #16157: What is the correct and *portable* way to do generic delegation? added

#2 - 10/15/2019 03:53 PM - Eregon (Benoit Daloze)

... has been proposed a few times as well, I'm not sure if there is a ticket for it:

```
def foo(...)
  @bar.foo(...)
```

#3 - 10/15/2019 05:20 PM - shevegen (Robert A. Heiler)

Hmm. I have not decided whether I like the proposal or not; I guess I am mostly

neutral, but with a slight tendency towards not being in favour of it. But leaving this aside, I think there are perhaps a few points of note.

1) Part of this proposal reminds me of delegate/delegation, e. g. delegating calls from one object to another - a bit like the Forwardable module may do. So a small issuer may be for other ruby users to understand the difference(s), towards the proposal here, and the forwardable module.

2) I think the core idea behind the proposal is primarily to save some keys, which on the one hand may be nice; on the other hand hmmm. To me personally, I do not understand why * would or could be used/retrofitted into meaning to "just pass all arguments". You also wrote that you are fine with other syntax; I believe that it may be better to see whether we could come up with another syntax altogether that still is short, could be used here, without adding a new meaning to *.

Benoit mentioned that there were other tickets for use of (...); I am not sure if there are other tickets for this specifically, but I recall having read that in other tickets, perhaps even proposed by matz (I don't remember, sorry).

I think using (...) would be a bit better than using/retrofitting *, even though * uses fewer characters. I am not a huge fan of (...) either though, but I do not dispute that it can be, in principle, useful. (Actually I just noticed that the link Benoit used pointed to your own suggestion. :D)

Personally I think getting the syntax "right" would be best. I am not sure how useful it would be, or how often it could be used; that might also have to be kept in mind. The recent python 3.8.0 release, for example, there was quite some discussion here and there about how useful or often used some of these features are. IMO whenever possible, the more people who CAN use a feature, and who also WILL use a feature, the better - so getting the syntax "right" here would be important.

I don't have a good proposal myself though.

I'll use a verbose dummy-example:

```
def foo(*bar)
  @some_other_object.foo(yield_arguments)
```

That's way too verbose, but I guess it may illustrate the goal of wanting to "yield the arguments onto that method". Actually we may have that already? Or perhaps not ... we have **method** ... perhaps we may need **arguments** too and then some syntactic sugar for it. It also reminds me a bit of "yield" and &proc - but anyway, IMO syntax matters. .foo(...) is a bit better than .foo(*) IMO, but not perfect. It may be difficult to get "great" results with very short syntax alone, without losing meaning.

And we'd even be future-proof if an eventual FOURTH kind of parameter is introduced!!!!

I don't think this is a good argument ;) because IF this were a problem, one could always suggest a proposal to see a change.

You can even find old joke-proposals such as:

```
def foo
  def bar
    def ble
  enddd
```

Or something like that. :P (Although I do have to admit that using several "end" can be a bit tedious; I just think that was a joke proposal since ... who in his sane mind wants to just spam the character "d", to mean end-of-scope, as several "end" would mean. Mandatory indent is also not that great either; I hate that I can't just copy/paste into the interactive python interpreter. IRB's behaviour is so much nicer and more convenient here.)

If rubyists must be told they have to change their forwarding code in 2.7 (due to keyword arguments), the pill might be easier to swallow if the change is a reduction rather than an increase in verbosity.

Well. I think matz said that this may be the only (or perhaps just one of the very few) changes between 2.x and 3.0, possibly the biggest one. I don't know the status about frozen Strings, but there are always those who like changes, and those who don't. Giving people time to prepare to switch is, IMO, always a good thing; it helps reduce problems in the long run. I don't think people are THAT opposed to change. But to come back to your comment - I don't think that the changes in regards to keyword arguments, should be connected to any syntax proposal in regards to delegation, for several reason. One is that I think they are not really that much connected; but also because the change in regards to keywords came, at the least partially, because it was confusing to many people. Matz even made jokes about it during some presentations.

I personally use oldschool hash options usually, barely any keyword arguments, so I am not affected really either way. Perhaps it would also be interesting to see what Jeremy thinks about syntax shortcuts/proposal in this regard, not solely confined to (*) or (...) but just in general.

#4 - 10/16/2019 10:33 AM - Eregon (Benoit Daloze)

Things to consider:

- Is * or ... an expression? What does `def m(...); a = ...; p a; end; m(1, a: 2) {}` print?
- Do we want to support required arguments before? It would be useful for `method_missing`:

```
def method_missing(name, ...)
  if name.to_s.end_with?('!')
    super
  else
    @target.send(name, ...)
  end
end
```

#5 - 10/16/2019 01:05 PM - Dan0042 (Daniel DeLorme)

- Is * or ... an expression? What does `def m(...); a = ...; p a; end; m(1, a: 2) {}` print?

I would tend to say `a = ...` is a syntax error; my intention was to use this only in the argument list of a method call, with an implementation similar to `super` without parentheses.

- Do we want to support required arguments before? It would be useful for `method_missing`

Then rather than "forward everything" the meaning would be more like "capture all extra arguments". That means we could have `foo(a, *)` or `foo(a, k:, *)` or `foo(a, k:, *, &b)` ... imho this is too complicated and it's better to just use the regular syntax at that point.

But it's true that often we want to operate on the arguments before forwarding, so I think maybe an asymmetric syntax like this would work best?

```
def method_missing(name, *) #currently valid syntax
  if name.to_s.end_with?('!')
    super
  else
    @target.send(***) #forward everything, including name
  end
end
```

#6 - 10/16/2019 01:17 PM - zverok (Victor Shepelev)

But it's true that often we want to operate on the arguments before forwarding, so I think maybe an asymmetric syntax like this would work best?

BTW, that's very valid point — similar problem was discussed here: <https://bugs.ruby-lang.org/issues/15049#change-73845>: There was a request for "all current method arguments list" API, and examples there were related to delegation, too. Quoting from there (my code example and comment)

```
def get(path:, accept: :json, headers: {}, **options)
  _request(method: :get, __all the rest of what have passed to this method__)
end

def post(path:, body:, accept: :json, headers: {}, **options)
  _request(method: :post, __all the rest of what have passed to this method__)
end
```

```
# ...and so on
```

Two of currently available options:

1. Accept just `**arguments`, and make checking what was mandatory, what should have default value and so on manually (also making auto-generated docs less expressive)
2. Accept everything as in my example, and then just do `_request(method: :get, path: path, body: body, accept: accept, headers: headers, **options)` ...that looks not DRY at all.

The solution proposed there was something like

```
def get(path:, accept: :json, headers: {}, **options)
  _request(method: :get, **kwargs) # pass ALL arguments
end
```

...but following your suggestion, it could've been

```
def get(path:, accept: :json, headers: {}, **options)
  _request(method: :get, ***) # pass ALL arguments
end
```

...which is kinda nice.

#7 - 10/16/2019 04:55 PM - jeremyevans0 (Jeremy Evans)

The disadvantage I see to this proposal is increased complexity. Both internal complexity in the implementation, and also more complexity for the user, as this adds more syntax Ruby programmers need to understand. However, I think the increased complexity for the user is probably offset by the fact that the ... syntax is simpler than `*a, **kw, &b` and probably more understandable for new Ruby programmers.

The main advantage I see to this proposal is potentially better performance (in CRuby). Currently, delegating using:

```
def foo(*a, **o, &b)
  @bar.foo(*a, **o, &b)
end
```

Causes an array allocation and multiple hash allocations for the delegation itself. Theoretically, delegating using:

```
def foo(...)
  @bar.foo(...)
end
```

should not cause any allocations for the delegation itself.

In terms of `*` vs. `...`, I would go with `...`. `@bar.foo(*)` doesn't imply to me that it would pass keyword arguments or a block, as `@bar.foo(*a)` wouldn't pass keyword arguments or a block.

I think there are some questions that need to be answered, if we decide to do this.

First, do we allow any other arguments in the method definition? If so, do we only allow mandatory positional arguments? Do we support optional positional arguments? I think it wouldn't make sense to support rest, keyword, or block arguments. Supporting mandatory positional arguments makes this more flexible and usable in more places, but also increases complexity.

Second, where we do allow ... when calling? Is this allowed and does it pass arguments from `foo` to `@bar.foo`?:

```
def foo(...)
  synchronize do |x|
    @bar.foo(...)
  end
end
```

Does this code pass arguments that `synchronize` yields to `@bar.foo`:

```
def foo(...)
  synchronize do |...|
    @bar.foo(...)
  end
end
```

Is this a `SyntaxError`?:

```
def foo
  @bar.foo(...)
end
```

What about:

```
PR = proc do
  @bar.foo(...)
end
def foo(...)
  instance_exec(&PR)
end
```

If ... can be implemented such that it improves performance over `*a, **kw, &b` (in CRuby), I think it may be worth adding. Otherwise, I don't think this is worth adding.

#8 - 10/16/2019 05:32 PM - Dan0042 (Daniel DeLorme)

Given the very interesting use case that zverok presented, I'm leaning more in favor of a lexically-scoped "operator" that doesn't need to be present in the method signature. So no invocation via block, just like `super`. Actually, the more it behaves similar to `super`, the easier it is to explain. So it would allow things like this:

```
def foo(a, b, c, d=1, e=2, f=3, g:10, h:11, i:12, j:false)
  super(42, ***) or @bar.foo(54, ***)
  #here, `super(***)` would be equivalent to `super`
end
```

(I've become partial to `***` because it looks like a splat plus a double splat, which is kind of what this shorthand means... it's a **hyper-splat!**)

#9 - 10/18/2019 11:17 PM - ioquatix (Samuel Williams)

Here are some real world examples from my code:

```
def self.for(*arguments, &block)
  self.new(block, *arguments)
end
```

Nicer?

```
def self.for(..., &block)
  self.new(block, ...)
end
```

Module to be prepended:

```
module Connection
  def initialize(*)
    super
```

```
    # Other stuff
  end
end
```

Nicer?

```
module Connection
  def initialize(...)
    super(...)
```

```
    # Other stuff
  end
end
```

Many repeated code:

```
def self.one(*arguments, **options)
  append One.new(*arguments, **options)
end
```

```
def self.many(*arguments, **options)
  append Many.new(*arguments, **options)
end
```

```
def self.split(*arguments, **options)
  append Split.new(*arguments, **options)
end
```

Nicer and more maintainable?

```
def self.split(...)
  append Split.new(...)
end
```

There are many more but since this feature is exciting to me, I wanted to give some specific use cases so we can evaluate how they would benefit/change.

#10 - 10/18/2019 11:33 PM - jeremyevans0 (Jeremy Evans)

ioquatix (Samuel Williams) wrote:

Here are some real world examples from my code:

```
def self.for(*arguments, &block)
  self.new(block, *arguments)
end
```

Nicer?

```
def self.for(..., &block)
  self.new(block, ...)
end
```

From reading the last dev meeting log (under Future work: lead argument handling is postponed), this will not be supported, at least initially.

Module to be prepended:

```
module Connection
  def initialize(*)
    super

```

```
    # Other stuff
  end
end
```

Nicer?

```
module Connection
  def initialize(...)
    super(...)

```

```
    # Other stuff
  end
end
```

I think a bare super makes more sense than super(...), and it is backwards compatible. However, in order to avoid keyword argument separation issues, if the super method accepts keyword arguments, you need to do def initialize(*, **) instead of def initialize(*) (def initialize(...) should also work).

Many repeated code:

```
def self.split(*arguments, **options)
  append Split.new(*arguments, **options)
end
```

Nicer and more maintainable?

```
def self.split(...)
  append Split.new(...)
end
```

Definitely looks nicer, so if you don't care about backwards compatibility, it seems like a good change.

#11 - 10/19/2019 02:56 AM - ioquatix (Samuel Williams)

The reason to support ... with other args is something like this:

```
class Controller < Container::Controller
  def initialize(command, *arguments, **options, &block)
    @command = command

    super(*arguments, **options, &block)
  end
end
```

```

    end
end

# Nicer?

class Controller < Container::Controller
  def initialize(command, ...)
    @command = command

    super(...)
  end
end

```

I think ... should be remainder of arguments that aren't explicitly consumed. Semantics might be a little bit more tricky to implement, but it makes a lot of sense to me and there are *many* places where such a syntax would make things not only clearer, but also faster by eliding allocations for **arguments* and ***options*.

#12 - 10/20/2019 11:56 AM - nobu (Nobuyoshi Nakada)

The parser itself was easy, but I'm wondering how ripper should treat it.

#13 - 10/20/2019 12:07 PM - nobu (Nobuyoshi Nakada)

ioquatix (Samuel Williams) wrote:

I think ... should be remainder of arguments that aren't explicitly consumed.

If it is the remainder, then it should be placed after all explicit arguments?

```
def foo(pre, opt = nil, *rest, kw:, &block, ...)
```

#14 - 10/21/2019 05:36 PM - nobu (Nobuyoshi Nakada)

- *Status changed from Open to Closed*

Applied in changeset [git|62d43828770211470bcacb9e943876f981b5a1b4](https://github.com/ruby/ruby/commit/62d43828770211470bcacb9e943876f981b5a1b4).

Arguments forwarding [Feature [#16253](#)]

#15 - 10/22/2019 07:37 PM - baweaver (Brandon Weaver)

Going to do a writeup on this later tonight if anyone wants to proof-read it, it'll be interesting to see what the wider community thinks but I really like it.

Also really loving the attention to detail Jeremy's been giving lately, really helps to clear up details.

#16 - 11/10/2019 10:15 AM - Eregon (Benoit Daloze)

- *Related to Feature #16296: Alternative behavior for `...` in method body if `...` is not in method definition added*

#17 - 11/10/2019 10:27 AM - Eregon (Benoit Daloze)

Note: this feature allows `def m(...)` but not `def m(meth, ...)` on current Ruby master.

I found that in some cases, the behavior is rather surprising as ... can also be the beginless endless Range:

```

$ ruby -e 'def m(...); p(...); end; m(1,2)'
1
2

$ ruby -e 'def m(...); p ...; end; m(1,2)'
^ nothing

$ ruby -e 'def m(...); p ...; end; p m(1,2)'
nil...nil

$ ruby -e 'def m(...); p(...[0]); end; m(1,2)'
...[0]

```

Can someone explain the second one?

I think we should clarify for this feature that ... isn't an object or an expression, it's only valid as arguments passed to a method.

#18 - 11/10/2019 10:39 AM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote:

Note: this feature allows `def m(...)` but not `def m(meth, ...)` on current Ruby master.

I found that in some cases, the behavior is rather surprising as `...` can also be the beginless endless Range:

```
$ ruby -e 'def m(...); p(...); end; m(1,2) '
1
2

$ ruby -e 'def m(...); p ...; end; m(1,2) '
^ nothing

$ ruby -e 'def m(...); p ...; end; p m(1,2) '
nil...nil

$ ruby -e 'def m(...); p(...[0]); end; m(1,2) '
...[0]
```

Can someone explain the second one?

It is parsed as an endless range `((p)...)...`.

#19 - 11/10/2019 09:23 PM - Eregon (Benoit Daloze)

- Status changed from Closed to Open

Is it intentional that this ticket was closed but `def m(meth, ...)` is a `SyntaxError`?

I'm going to reopen this, because I think it is severely limited for delegation otherwise. For example, it can't be used in

```
def method_missing(name, ...)
  if name.to_s.end_with?('=')
    update(name, ...)
  else
    # ...
  end
end
```

Using a helper method would be one way, but it's quite ugly:

```
def first_arg(*args)
  args.first
end

def method_missing(...)
  name = first_arg(...)
  if name.to_s.end_with?('=')
    update(...)
  else
    # ...
  end
end
```

And would quickly become unfeasible if, for instance, the delegated method doesn't take the name argument, or not as first argument.

#20 - 11/10/2019 09:41 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

Is it intentional that this ticket was closed but `def m(meth, ...)` is a `SyntaxError`?

This is expected at present. Lead argument handling will probably happen in the future. From the notes of the last dev meeting:

```
Future work: lead argument handling is postponed
* lead arguments can be extracted
* lead arguments can be added
* def f(x, y, ...); g(1, 2, ...); end
```


It is true that this means the syntax only handles a subset of delegation methods. You can always do things the longer way if you need more control:

```
ruby2_keywords def method_missing(name, *args)
  if name.to_s.end_with?('=')
    update(name, *args)
  else
    # ...
  end
end
end
```

#21 - 11/10/2019 10:57 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

It is true that this means the syntax only handles a subset of delegation methods. You can always do things the longer way if you need more control:

Right, except if one wants that code to work on Ruby 2.7+.
I think ... could be part of how to do delegation right in the future, succinctly:
<https://eregon.me/blog/2019/11/10/the-delegation-challenge-of-ruby27.html>

I think leading required arguments are the most most needed in delegation.

#22 - 11/12/2019 08:05 AM - mame (Yusuke Endoh)

- Status changed from Open to Closed

Applied in changeset [git|d1ae2bc27fd4183e6abb9e83691e192bfe1e5316](https://github.com/ruby/ruby/commit/d1ae2bc27fd4183e6abb9e83691e192bfe1e5316).

NEWS: Make it clear that delegation syntax (...) requires parentheses

Ref [Feature [#16253](#)]

#23 - 11/27/2019 04:50 PM - Eregon (Benoit Daloze)

- Related to Feature [#16378](#): Support leading arguments together with ... added

#24 - 11/27/2019 04:51 PM - Eregon (Benoit Daloze)

I filed [#16378](#) for supporting leading arguments with