since capturing a block into a proc is slow: foo(&block)
and creating chains of blocks is kind of ugly and ultimately also inefficient: foo{yield}
why not allow block forwarding without capturing: foo(&) foo(1, 2, &)

Related issues:
- Related to Ruby master - Feature #3447: argument delegation
  Closed
- Related to Ruby master - Feature #14045: Lazy Proc allocation for block param...
  Closed

Associated revisions
Revision 4adb0129 - 11/18/2021 10:17 PM - jeremyevans (Jeremy Evans)
Anonymous block forwarding allows a method to forward a passed
block to another method without having to provide a name for the
block parameter.

Implements [Feature #11256]
Co-authored-by: Yusuke Endoh mame@ruby-lang.org
Co-authored-by: Nobuyoshi Nakada nobu@ruby-lang.org

History
#1 - 06/13/2015 06:44 PM - bughit (bug hit)

# takes a block
def bar
  # forwards it to foo without instantiating a proc
  foo({1, &})
end

#2 - 06/22/2015 07:44 AM - matz (Yukihiro Matsumoto)
- Related to Feature #3447: argument delegation added

#3 - 11/27/2017 11:50 AM - mame (Yusuke Endoh)
- Related to Feature #14045: Lazy Proc allocation for block parameters added

#4 - 11/27/2017 11:55 AM - mame (Yusuke Endoh)
- Assignee set to matz (Yukihiro Matsumoto)
- Status changed from Open to Assigned

Lazy Proc allocation for block parameters (#14045) is implemented, so capturing a block into a proc is not slow.

One of the motivation now disappeared, but there is another motivation of this feature: the simplicity of the notation. Matz, is this syntax still hopeful to be accepted?

#5 - 11/29/2017 09:45 AM - matz (Yukihiro Matsumoto)
- Status changed from Assigned to Closed

As mame (Yusuke Endoh) stated, we don't need it after we had lazy Proc allocation.

Matz.

#6 - 12/04/2017 07:49 AM - ko1 (Koichi Sasada)
but there is another motivation of this feature: the simplicity of the notation

With proposal syntax, we don't need to use a variable name for block parameter. What do you think about it?

---

#7 - 12/12/2017 08:34 AM - matz (Yukihiro Matsumoto)
Accepted.

Matz.

#8 - 12/12/2017 09:45 AM - nobu (Nobuyoshi Nakada)
Just to be clear, does it require both of the definition and the use, instead of only the latter?

I mean this is allowed:

```ruby
def foo()
  bar()
end
```

but these are not:

```ruby
def foo()
  bar(&)
end

def foo(&block)
  bar(&)
end
```

---

#9 - 12/12/2017 10:04 AM - nobu (Nobuyoshi Nakada)
Current patch.

diff --git a/parse.y b/parse.y
index 1026d5c896..2a98016002 100644
--- a/parse.y
+++ b/parse.y
@@ -347,6 +347,8 @@ static int parser_yyerror(struct parser_params*, const char*);
#define lambda_beginning_p() (lpar_beg && lpar_beg == paren_nest)
+#define ANON_BLOCK_ID '&'
+
static enum yytokentype yylex(YYSTYPE*, YYLTYPE*, struct parser_params*);

#ifndef RIPPER
@@ -2560,6 +2562,18 @@ block_arg : tAMPER arg_value
  $$ = $2;
  %*/
  |
+  | tAMPER
+  +
+  + /**%*/
+  +  if ((local_id(ANON_BLOCK_ID)) {
+  + compile_error(PARSER_ARG "no anonymous block parameter");
+  +  }
+  +  $$ = NEW_BLOCK_PASS(new_lvar(ANON_BLOCK_ID, &@1));
+  + $$->nd_loc = $$;
+  + /%*/
+  + $$ = Qnil;
+  + %*/
+  + %*/
  + ;

opt_block_arg : ',' block_arg
@@ -4913,6 +4927,15 @@ f_block_arg : blkarg_mark tIDENTIFIER
  $$ = dispatch1(blockarg, $2);
  %*/
  |
```
Just to be clear, does it require both of the definition and the use, instead of only the latter?

I mean this is allowed:

```ruby
def foo(&)
  bar(&)
end
```

but these are not:

```ruby
def foo()
  bar(&)
end
```

```ruby
def foo(&block)
  bar(&)
end
```

def foo(&) is more self documenting so should be legal syntax, but should not be required because in general a method taking a block does not have to be marked.
- (VALUE val) // inc += (int)(ci->orig_argc + ((ci->flag & VM_CALL_ARGS_BLOCKARG) ? 1 : 0));
+(VALUE val) // inc += (int)(ci->orig_argc + (((ci->flag & (VM_CALL_ARGS_BLOCKARG | VM_CALL_ARGS_BLOCKARG_THROUGH)) == VM_CALL_ARGS_BLOCKARG) ? 1 : 0));
{
    struct rb_calling_info calling;
@@ -924,7 +924,7 @@ DEFINE_INSN
    invokesuper
    (CALL_INFO ci, CALL_CACHE cc, ISEQ blockiseq)
    (...)
- (VALUE val) // inc += (int)(ci->orig_argc + ((ci->flag & VM_CALL_ARGS_BLOCKARG) ? 1 : 0));
+ (VALUE val) // inc += (int)(ci->orig_argc + (((ci->flag & (VM_CALL_ARGS_BLOCKARG | VM_CALL_ARGS_BLOCKARG_THROUGH)) == VM_CALL_ARGS_BLOCKARG) ? 1 : 0));
    {
        struct rb_calling_info calling;
        calling.argc = ci->orig_argc;
        ...
Just my opinion, but I think the shortcut syntax is going to cause more confusion than it would help. & as argument without a & as a parameter looks very weird to me (it's like it in other languages, I would strongly advise against such magic). I argue adding this makes Ruby less readable as a language.

This only saves one character in the definition, and it's not hard to name a block: &b or &block, that's what the Ruby code out there uses and is clear because of it. Moreover, it's inconsistent with arguments which must be named to be passed (except via zsuper): foo(*) does not work as a call, and def meth(*) means ignore arguments, which is meaningless for blocks (just omitting &b is enough to ignore a block).

This also seems to conflict with one of the nicer proposals for short block notations like enum.map(&.to_s.upcase.ljust(3)), a much more general and useful feature than this shortcut, which only applies for a few methods forwarding the block and saves 2 character per forwarding method.

Also, the original poster started with "since capturing a block into a proc is slow: foo(&block)". There is no such reason anymore, and I see no clear motivation of why such a special edge-case syntax is worth adding.
From the developer meeting notes, [Feature #11256] anonymous block forwarding:
Matz: it's a good property to avoid naming variables. Also I personally like it. However prompting such name-less programming might let people write cryptic codes.
Matz: hmm… Made up my mind. Accepted.

I agree it is cryptic and moreover argue it has little use, is inconsistent with other arguments and likely to cause syntax restrictions for other features.

bughit (bug hit) Could you explain your motivation for this shortcut now that the performance is no longer a concern?
I agree foo[yield] was not nice, but is it worth to have foo(&) foo(1, 2, &) over foo(&b) foo(1, 2, &b) ?

matz (Yukihiro Matsumoto) Could you detail your opinion and reconsider?

#13 - 12/14/2017 04:56 PM - bughit (bug hit)
Eregon (Benoit Daloze) wrote:

bughit (bug hit) Could you explain your motivation for this shortcut now that the performance is no longer a concern?
I agree foo[yield] was not nice, but is it worth to have foo(&) foo(1, 2, &) over foo(&b) foo(1, 2, &b) ?

ko1 convinced matz to accept this and he provided his reasons, simplicity of notation, not requiring an otherwise pointless variable. Which makes sense to me.

Also keep in mind that lazy proc allocation is an implementation detail. Conceptually when you declare a block params it still looks like you're doing unnecessary work of instantiating a proc object which you have no intention of using.

Every method has an invisible, nameless, optional block, and a naked & seems like an intuitive, logical way to forward it.

#14 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)
- Target version deleted (2.6)

#15 - 10/28/2021 10:01 PM - jeremyevans0 (Jeremy Evans)
Looks like matz (Yukihiro Matsumoto) approved this, but it was never committed. I've submitted a pull request for this that is based on the patches provided by mame (Yusuke Endoh) and nobu (Nobuyoshi Nakada): https://github.com/ruby/ruby/pull/5051

#16 - 11/18/2021 02:00 PM - mame (Yusuke Endoh)
matz (Yukihiro Matsumoto) accepted this, but said that the formal argument must explicitly receive a anonymous block parameter, i.e.

def foo(&) = bar(&)  # OK
def foo = bar(&)    # NG

In the future, matz wants to make it more explicit whether a method accepts a block or not.

jeremyevans0 (Jeremy Evans) Could you remove my diff from your PR?

#17 - 11/18/2021 08:55 PM - jeremyevans0 (Jeremy Evans)
mame (Yusuke Endoh) wrote in #note-16:

matz (Yukihiro Matsumoto) accepted this, but said that the formal argument must explicitly receive a anonymous block parameter, i.e.

def foo(&) = bar(&)  # OK
def foo = bar(&)    # NG

In the future, matz wants to make it more explicit whether a method accepts a block or not.

jeremyevans0 (Jeremy Evans) Could you remove my diff from your PR?

Sure, I updated the pull request. This made the pull request much simpler: https://github.com/ruby/ruby/pull/5051

#18 - 11/18/2021 09:59 PM - bughit (bug hit)
mame (Yusuke Endoh) wrote in #note-16:

matz (Yukihiro Matsumoto) accepted this, but said that the formal argument must explicitly receive a anonymous block parameter, i.e.

def foo(&) = bar(&)  # OK
def foo = bar(&)    # NG

In the future, matz wants to make it more explicit whether a method accepts a block or not.
Where is the naked ampersand in def foo(&) described?

Don't see it here: https://docs.ruby-lang.org/en/master/doc/syntax/methods_rdoc.html

Does it mean the block is required?

What is the reason for the restriction?

The block may or may not be passed to def foo, so what is wrong with forwarding it or its absence with &? It reduces the utility of this feature, for what gain?

#19 - 11/18/2021 10:12 PM - jeremyevans0 (Jeremy Evans)
bughit (bug hit) wrote in #note-18:

mame (Yusuke Endoh) wrote in #note-16:

matz (Yukihiro Matsumoto) accepted this, but said that the formal argument must explicitly receive a anonymous block parameter, i.e.

```ruby
def foo(&) = bar(&)  # OK
def foo = bar(&)     # NG
```

In the future, matz wants to make it more explicit whether a method accepts a block or not.

Where is the naked ampersand in def foo(&) described?

Don't see it here: https://docs.ruby-lang.org/en/master/doc/syntax/methods_rdoc.html

This hasn't been committed yet, so you wouldn't see it on the website yet. The pull request already includes updates to the documentation: https://github.com/ruby/ruby/pull/5051/files#diff-d2caf8d6be0d2ccf90906e4f9bd1b4e4738586b4d5332954d57640e6fca0fd88

Does it mean the block is required?

No, just as &block doesn't mean the block is required.

What is the reason for the restriction?

The block may or may not be passed to def foo, so what is wrong with forwarding it or its absence with &? It reduces the utility of this feature, for what gain?

In the future, matz wants to make it more explicit whether a method accepts a block or not. I assume that means that we could see future changes in Ruby that better align with requiring & as a parameter, even if currently it seems reasonable that it would not be required.

In terms of gain currently, if & is required, the code to implement the feature is much simpler. Ruby is already very complex internally. If we can reduce the amount of complexity added, that is a good reason.

#20 - 11/18/2021 10:18 PM - jeremyevans (Jeremy Evans)
- Status changed from Assigned to Closed

Applied in changeset giti4adb012926f8bd6011168327d8832cf19976de40.

Anonymous block forwarding allows a method to forward a passed block to another method without having to provide a name for the block parameter.

Implements [Feature #11256]

Co-authored-by: Yusuke Endoh mame@ruby-lang.org
Co-authored-by: Nobuyoshi Nakada nobu@ruby-lang.org

#21 - 11/18/2021 10:35 PM - bughit (bug hit)
jeremyevans0 (Jeremy Evans) wrote in #note-19:

Does it mean the block is required?

No, just as &block doesn't mean the block is required.
&block serves a purpose in the code, binding the block to a named proc.

A naked & is just documentation then. It has not material effect. It makes no sense to force such a noop modifications of method signatures to be able to forward blocks anonymously. If the block can be present it should be forwardable.

What is the reason for the restriction?

The block may or may not be passed to def foo, so what is wrong with forwarding it or its absence with &? It reduces the utility of this feature, for what gain?

In the future, matz wants to make it more explicit whether a method accepts a block or not.

That only makes sense if def foo comes to mean the block is not allowed. It seems doubtful that will ever happen and until then anon forwarding should be possible without & in the signature.

If we can reduce the amount of complexity added, that is a good reason.

The result of this complexity reduction is an incoherent restriction that cannot be justified logically. Considering that the more complex proper implementation already exists, it's worth keeping it.

#22 - 11/19/2021 07:54 PM - Eregon (Benoit Daloze)
bughit (bug hit) wrote in #note-21:

A naked & is just documentation then. It has not material effect. It makes no sense to force such a noop modifications of method signatures to be able to forward blocks anonymously. If the block can be present it should be forwardable.

It's extremely useful documentation/reading help that the method accepts a block.

If people have to scan for & in the whole method to know if a method accepts a block, it's really annoying (a yield keyword is much easier to spot). I strongly agree with matz on this, methods taking a block should be clear about it, including in the parameters.

The result of this complexity reduction is an incoherent restriction that cannot be justified logically. Considering that the more complex proper implementation already exists, it's worth keeping it.

matz said no.
Please have some understanding that people implementing Ruby might know better about this and complexity.

#23 - 11/19/2021 08:48 PM - bughit (bug hit)

I strongly agree with matz on this, methods taking a block should be clear about it, including in the parameters.

I never said I disagree. Requiring a naked & in the signature to signal "block allowed" and perhaps &! for block required would be better. But "block allowed" will break lots of code so I don't see it happening. So as long as "block allowed" is the default, forwarding should be possible. ... forwards blocks with 0 indication that the method takes a block, should it be illegal without a naked & in the signature?

Please have some understanding that people implementing Ruby might know better about this and complexity.

This is my feature request, so please don't tell me not to advocate for my vision.

#24 - 11/21/2021 04:26 PM - mame (Yusuke Endoh)
bughit (bug hit) wrote in #note-21:

That only makes sense if def foo comes to mean the block is not allowed.

From what I understand, matz is for this. Passing a block to def foo might be prohibited in the future (I guess it is very far even if possible, though). If a method receives a block, it should have a &block parameter or ... argument forwarding. For the compatibility, a method whose body has yield or super keywords might be also allowed.

Actually ko1 (Koichi Sasada) tried it in #15554. He withdrew the proposal because he have discovered a code pattern that makes false positives inevitable, but if matz accepts the incompatibility, the proposal might be a good first step.

Matz said he will create a ticket to discuss it, so please wait for him about the details.