

Ruby master - Feature #16461

Proc#using

12/28/2019 03:32 AM - shugo (Shugo Maeda)

Status:	Assigned
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	3.0

Description

Overview

I propose Proc#using to support block-level refinements.

```
module IntegerDivExt
  refine Integer do
    def /(other)
      quo(other)
    end
  end
end

def instance_eval_with_integer_div_ext(obj, &block)
  block.using(IntegerDivExt) # using IntegerDivExt in the block represented by the Proc object
  obj.instance_eval(&block)
end

# necessary where blocks are defined (not where Proc#using is called)
using Proc::Refinements

p 1 / 2 #=> 0
instance_eval_with_integer_div_ext(1) do
  p self / 2 #=> (1/2)
end
p 1 / 2 #=> 0
```

PoC implementation

For CRuby: <https://github.com/shugo/ruby/pull/2>

For JRuby: <https://github.com/shugo/jruby/pull/1>

Background

I proposed [Feature #12086: using: option for instance_eval etc.](#) before, but it has problems:

- Thread safety: The same block can be invoked with different refinements in multiple threads, so it's hard to implement method caching.
- `_exec` family support: `{instance,class,module}_exec` cannot be supported.
- Implicit use of refinements: every blocks can be used with refinements, so there was implementation difficulty in JRuby and it has usability issue in headius's opinion.

Solutions in this proposal

Thread safety

Proc#using affects the block represented by the Proc object, neither the specific Proc object nor the specific block invocation. Method calls in a block are resolved with refinements which are used by Proc#using in the block at the time. Once all possible refinements are used in the block, there is no need to invalidate method cache anymore.

See [these tests](#) to understand how it works.

Which refinements are used is depending on the order of Proc#using invocations until all Proc#using calls are finished, but eventually method calls in a block are resolved with the same refinements.

* `_exec` family support

[Feature #12086](#) was an extension of `_eval` family, so it cannot be used with `_exec` family, but `Proc#using` is independent from `_eval` family, and can be used with `_exec` family:

```
def instance_exec_with_integer_div_ext(obj, *args, &block)
  block.using(IntegerDivExt)
  obj.instance_exec(*args, &block)
end

using Proc::Refinements

p 1 / 2 #=> 0
instance_exec_with_integer_div_ext(1, 2) do |other|
  p self / other #=> (1/2)
end
p 1 / 2 #=> 0
```

Implicit use of refinements

`Proc#using` can be used only if `using Proc::Refinements` is called in the scope of the block represented by the `Proc` object. Otherwise, a `RuntimeError` is raised.

There are two reasons:

- JRuby creates a special `CallSite` for refinements at compile-time only when `using` is called at the scope.
- When reading programs, it may help understanding behavior. IMHO, it may be unnecessary if libraries which uses `Proc#using` are well documented.

`Proc::Refinements` is a dummy module, and has no actual refinements.

Related issues:

Related to Ruby master - Feature #12086: `using`: option for `instance_eval` etc.

[Open](#)

History

#1 - 12/29/2019 06:30 PM - Eregon (Benoit Daloze)

This still has the problem that it mutates the `Proc`, what should happen if another Thread concurrently does `block.using(OtherRefinement)` ?

Also it still seems inefficient, at least if there are `block.using(refinement)` with different refinements. IMHO refinements should remain lexically scoped, so for a given call site it always means the same set of refinements.

That's how it's implemented in TruffleRuby, we use the guarantee that at a given call site refinements cannot change. So then we can just consider the used refinements during the first method lookup, and after don't need to check anything extra, which mean zero overhead for refinements on peak performance. There is no special detection for `using`, every call site considers refinements during method lookup.

For your example above, I think using `IntegerDivExt` at the top level would be much clearer. Do you have a motivating example where that approach is much better than existing refinements?

#2 - 12/29/2019 07:43 PM - Eregon (Benoit Daloze)

- Related to Feature #12086: `using`: option for `instance_eval` etc. added

#3 - 12/31/2019 02:09 AM - shugo (Shugo Maeda)

Eregon (Benoit Daloze) wrote:

This still has the problem that it mutates the `Proc`, what should happen if another Thread concurrently does `block.using(OtherRefinement)` ?

It doesn't mutate the `Proc`, but the block, and if `OtherRefinement` is activated before the block invocation, both refinements are activated. If the refinements are conflicted, it doesn't work, but I don't think such a situation happens in real world use cases.

Also it still seems inefficient, at least if there are `block.using(refinement)` with different refinements. IMHO refinements should remain lexically scoped, so for a given call site it always means the same set of refinements.

That's how it's implemented in TruffleRuby, we use the guarantee that at a given call site refinements cannot change. So then we can just consider the used refinements during the first method lookup, and after don't need to check anything extra, which mean zero overhead for refinements on peak performance.

There is no special detection for using, every call site considers refinements during method lookup.

Is it hard to invalidate cache only when new refinements are activated by Proc#using?

In my proposal refinements are activated per block (not per Proc), so the refinements activated in a given call site are eventually same.

Or it may be possible to prohibit adding *new* refinements after the first call of the block.

For your example above, I think using IntegerDivExt at the top level would be much clearer.

Do you have a motivating example where that approach is much better than existing refinements?

using IntegerDivExt activates refinements in that entire scope, but I'd like to narrow the scope to blocks for DSLs which refine built-in classes (e.g, <https://github.com/amatsuda/activerecord-refinements>).

Allowing using in blocks may be used instead, but it's too verbose for such DSLs.

#4 - 01/02/2020 12:44 PM - Eregon (Benoit Daloze)

shugo (Shugo Maeda) wrote:

It doesn't mutate the Proc, but the block, and if OtherRefinement is activated before the block invocation, both refinements are activated. If the refinements are conflicted, it doesn't work, but I don't think such a situation happens in real world use cases.

So what happens if I have this:

```
using Proc::Refinements
```

```
block = Proc.new do
  1 / 2
  3 + 4
end
```

```
Thread.new { block.using(DivRefinement) }
Thread.new { block.using(AddRefinement) }
block.call
```

Can I have / be the original one (Integer#/) and + be the refined one (AddRefinement#+) in the same block.call?

In my proposal refinements are activated per block (not per Proc), so the refinements activated in a given call site are eventually same.

I start to understand what you mean by that, block.using(IntegerDivExt) means every Proc instance of that block will use those refinements, so:

```
using Proc::Refinements
```

```
def my_proc
  Proc.new { 1 / 2 }
end
```

```
proc = my_proc
proc.using(IntegerDivExt)
proc.call # Uses IntegerDivExt
```

```
my_proc.call # Uses IntegerDivExt too even though it's not the same Proc object
```

So Proc#using mutates the refinements on that block, globally, for all future invocations of that block.

I think this will be very hard to explain and document, unfortunately.

Is it hard to invalidate cache only when new refinements are activated by Proc#using?

Basically this slows down call sites inline caches, by having to always check the activated refinements of that block.

Or, we could invalidate all method lookups inside that block whenever Proc#using is used with a new refinement, and then we wouldn't need to check it.

How would this work with Guilds or MVM?

Would it imply having a copy of the block bytecode per Guild/VM to have inline caches for call sites inside per Guild/VM?

Or Proc#using would affect the calls in other Guilds too? That seems unacceptable for the MVM case at least.

using IntegerDivExt activates refinements in that entire scope, but I'd like to narrow the scope to blocks for DSLs which refine built-in classes (e.g, <https://github.com/amatsuda/activerecord-refinements>).

I see, you want

```
using Proc::Refinements

User.where { :age > 3 }
User.where { :name == 'matz' }
```

to work with Symbol#> and Symbol#== refinements?

Overriding Symbol#== in that block does seem pretty dangerous to me.
Any == in that block suddenly becomes a hard-to-diagnose bug.
Example:

```
using Proc::Refinements

mode = get_current_mode
allowed = User.where { mode == :admin ? :name == 'matz' : :age > 3 }
```

Maybe not being able to have refinements per block encourages people to not refine Integer#> and Symbol#==, which might be a lot safer anyway. Refining existing methods feels quite dangerous to me in general. Refining new methods seems fine and safe.

I understand the intent, and not willing to have using ActiveRecord::DSL at the top-level as it would then override Symbol#== everywhere in that file.

Or it may be possible to prohibit adding *new* refinements after the first call of the block.

I think that could be a good rule, all Proc#using must happen before Proc#call for a given block. That would mean the code inside a block always calls the same methods (i.e., the refinements for a given call site are fixed), which seems essential to me for sanity when reading code.

I think something like this should be prevented if feasible:

```
using Proc::Refinements

b = Proc.new { :age > 3 }

b.call # no refinements
User.where(&b) # using refinements
b.call # using refinements because Proc#using is global state!
```

It would be nice if we could kind of mark methods taking a *lexical* block (where {}, not where(&b)) and wanting to apply refinements to them:

```
class ActiveRecord::Model
  def where
    yield
  end
  use_refinements_for_method :where, DSLRefinements
end
```

Which might make it possible to require that where only accepts a lexical block. I'm thinking all where(&b) are going to be confusing, because the block body is far from its usage which enables refinements.

Allowing using in blocks may be used instead, but it's too verbose for such DSLs.

Isn't using Proc::Refinements already too verbose for Rails users?

It would be useful as a hint we need to be able to invalidate all lookups in blocks affected by using Proc::Refinements though. Otherwise we would need the ability to invalidate every block, which means in TruffleRuby a higher memory footprint of one Assumption for every block. OTOH, using Proc::Refinements would likely apply to far more blocks than needed, if e.g., used at the top-level of the file. Imagine using Proc::Refinements at the top of https://github.com/amatsuda/activerecord-refinements/blob/master/spec/where_spec.rb for example.

#5 - 01/07/2020 01:39 AM - shugo (Shugo Maeda)

Eregon (Benoit Daloz) wrote:

shugo (Shugo Maeda) wrote:

It doesn't mutate the Proc, but the block, and if OtherRefinement is activated before the block invocation, both refinements are activated. If the refinements are conflicted, it doesn't work, but I don't think such a situation happens in real world use cases.

So what happens if I have this:

```
using Proc::Refinements

block = Proc.new do
  1 / 2
  3 + 4
end

Thread.new { block.using(DivRefinement) }
Thread.new { block.using(AddRefinement) }
block.call
```

Can I have / be the original one (Integer#/) and + be the refined one (AddRefinement#+) in the same block.call?

No, you can't have / be the original one once block.using(DivRefinement) is called.

I don't come up with use cases using conflicting refinements in the same block, so it may be better to prohibit activating new refinements after the first block call.

So Proc#using mutates the refinements on that block, globally, for all future invocations of that block.

Yes.

I think this will be very hard to explain and document, unfortunately.

I admit that the behavior is complex, but DSL users need not understand details.

Is it hard to invalidate cache only when new refinements are activated by Proc#using?

Basically this slows down call sites inline caches, by having to always check the activated refinements of that block.

Or, we could invalidate all method lookups inside that block whenever Proc#using is used with a new refinement, and then we wouldn't need to check it.

I meant the latter, but it's not necessary if we prohibit activating new refinements after the first block call.

How would this work with Guilds or MVM?

Would it imply having a copy of the block bytecode per Guild/VM to have inline caches for call sites inside per Guild/VM?

Or Proc#using would affect the calls in other Guilds too? That seems unacceptable for the MVM case at least.

I have no idea with Guilds, but I agree with you as for MVM.

using IntegerDivExt activates refinements in that entire scope, but I'd like to narrow the scope to blocks for DSLs which refine built-in classes (e.g. <https://github.com/amatsuda/activerecord-refinements>).

I see, you want

```
using Proc::Refinements

User.where { :age > 3 }
User.where { :name == 'matz' }
```

to work with Symbol#> and Symbol#== refinements?

Yes.

Overriding Symbol#== in that block does seem pretty dangerous to me.

Any == in that block suddenly becomes a hard-to-diagnose bug.

Example:

```
using Proc::Refinements

mode = get_current_mode
allowed = User.where { mode == :admin ? :name == 'matz' : :age > 3 }
```

Maybe not being able to have refinements per block encourages people to not refine Integer#/ and Symbol#==, which might be a lot safer

anyway.

Refining existing methods feels quite dangerous to me in general. Refining new methods seems fine and safe.

I agree with you in general.

There is a trade off between power and safety, but I think the current specification is too conservative.

I understand the intent, and not willing to have using ActiveRecord::DSL at the top-level as it would then override Symbol#== everywhere in that file.

Yes.

Or it may be possible to prohibit adding *new* refinements after the first call of the block.

I think that could be a good rule, all Proc#using must happen before Proc#call for a given block.

That would mean the code inside a block always calls the same methods (i.e., the refinements for a given call site are fixed), which seems essential to me for sanity when reading code.

OK, I'll try to fix PoC implementation.

It would be nice if we could kind of mark methods taking a *lexical* block (where {}, not where(&b)) and wanting to apply refinements to them:

```
class ActiveRecord::Model
  def where
    yield
  end
  use_refinements_for_method :where, DSLRefinements
end
```

Which might make it possible to require that where only accepts a lexical block.

I'm thinking all where(&b) are going to be confusing, because the block body is far from its usage which enables refinements.

It's an interesting idea, but I heard a discussion about deprecating yield from CRuby committers before.

Does anyone know the status of the discussion?

And I want to provide a way to use refinements with instance_eval.

Allowing using in blocks may be used instead, but it's too verbose for such DSLs.

Isn't using Proc::Refinements already too verbose for Rails users?

Maybe. I proposed it mainly for JRuby implementation.

It would be useful as a hint we need to be able to invalidate all lookups in blocks affected by using Proc::Refinements though.

Otherwise we would need the ability to invalidate every block, which means in TruffleRuby a higher memory footprint of one Assumption for every block.

OTOH, using Proc::Refinements would likely apply to far more blocks than needed, if e.g., used at the top-level of the file.

Imagine using Proc::Refinements at the top of https://github.com/amatsuda/activerecord-refinements/blob/master/spec/where_spec.rb for example.

I understand your concern.

Magic comments like block-level-refinements: true may be used instead.

#6 - 08/08/2020 01:29 AM - shugo (Shugo Maeda)

- Assignee set to matz (Yukihiko Matsumoto)

- Status changed from Open to Assigned

shugo (Shugo Maeda) wrote in [#note-5](#):

I think that could be a good rule, all Proc#using must happen before Proc#call for a given block.

That would mean the code inside a block always calls the same methods (i.e., the refinements for a given call site are fixed), which seems essential to me for sanity when reading code.

OK, I'll try to fix PoC implementation.

Sorry for the delay.

I've fixed the CRuby implementation:

<https://github.com/shugo/ruby/pull/2>

#7 - 09/05/2020 11:05 AM - palkan (Vladimir Dementyev)

Disclaimer: I'm a big fan of refinements; they make Ruby more expressive (or chaotic ☹️) and allow developers to be more creative.

shugo (Shugo Maeda) wrote in [#note-5](#):

I admit that the behavior is complex, but DSL users need not understand details.

Unless they have to debug the code using this DSL, understanding the details leads to better code; IMO, Rails has enough magic to blow newcomers' minds.

At the same time, Refinements are already too complex for most developers, even if we only deal with lexical scope and explicit activation. With Proc#using, we will likely make Refinements a secret knowledge only available to magicians.

I understand the intent, and not willing to have using ActiveRecord::DSL at the top-level as it would then override Symbol#== everywhere in that file.

What about introducing unusing or similar instead?

```
class User < ActiveRecord::Base
  using ActiveRecord::WhereDSL

  scope :not_so_young, -> { where { :age >= 30 } }

  # deactivate refinement below this line
  unusing ActiveRecord::WhereDSL
end
```

Looks a bit uglier but, IMO, much easier to understand for end-users, especially if they're familiar with refinements: we still use lexical scope, nothing new here.

Another idea is to pass block to using (so, it's gonna be a mix of instance_eval + Proc#using, I think):

```
using ActiveRecord::WhereDSL do
  scope :not_so_young, -> { where { :age >= 30 } }
end
```

#8 - 09/05/2020 11:49 AM - Eregon (Benoit Daloze)

I think it would be good to get [headius \(Charles Nutter\)](#)' feedback on this.

I'd also like to take another detailed look (I won't be able before 14 September though).

#9 - 09/29/2020 03:37 AM - hsbt (Hiroshi SHIBATA)

- Target version changed from 36 to 3.0

#10 - 10/22/2020 10:58 AM - Eregon (Benoit Daloze)

Reading [#12086](#) again, I feel [#12281](#) is a much simpler alternative (using(refinement) { literal block }). Wouldn't [#12281](#) be enough?

block.using(IntegerDivExt) is mutating state and this feels too magic and dangerous to me.

I think DSL users can accept to wrap their DSL code with using(refinement) { ... } (or using refinement in a module/file).

That's actually not more verbose than using Proc::Refinements and I think so much clearer.

We could have additional requirements to ensure it's truly lexical and the refinement given to a given using(refinement) { ... } call site is always the same.

I think that would be good, because then we know code inside always use the given refinements, much like how using refinement works today.

For the User.where { :age > 3 } case, if we want to avoid an extra using(refinement) { ... }, I think it could be fine to automatically enable refinements in that block if we can ensure it's still fully lexical.

Concretely that would mean for that block { :age > 3 } it's always called with the same refinements, no matter where it's called from.

Attempting to use any other refinement would raise an error.

Maybe a way to design this restriction could be to annotate the method where with "enables MyRefinement on the lexical block given to it".

```
class ModelClassMethods
  def where(&block)
    ...
    block.call / yield
  end
  instance_method(:where).use_refinements_for_block(ActiveRecord::WhereDSL)
end
```

```
User.where { :age > 3 }
```

That way I think we can ensure the given literal block always has the exact same refinements.

Multiple use_refinements_for_block calls for a method would be an error.

And use_refinements_for_block should be used before the method is called (so there is no block before executed without refinements).

Implementation-wise, where would modify the refinements for the block/Proc, before executing the body of where, so it would all be transparent to the user.

#11 - 10/22/2020 01:07 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloz) wrote in [#note-10](#):

Maybe a way to design this restriction could be to annotate the method where with "enables MyRefinement on the lexical block given to it".

I like this idea. Since the block should always have the same refinements, it makes sense to define them once rather than every time the block is called. How about

```
class ModelClassMethods
  using ActiveRecord::WhereDSL, def where(&block)
    ...
    block.call / yield
  end
end
```