Ruby master - Feature #12901

Anonymous functions without scope lookup overhead

11/04/2016 04:54 PM - schneems (Richard Schneeman)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Right now if you are writing performance sensitive code you can get a performance bump by not using anonymous functions (procs, lambdas, blocks) due to the scope lookup overhead.

https://twitter.com/apotonick/status/791167782532493312?lang=en

I would like to be able to create an anonymous function and specify I don't need it to perform the lookup when called.

I am thinking that this code:

```
Proc.new(scope: false) {|var| puts var }
```

Would be the equivalent of

```
def anon(var)
  puts var
end
```

If you call it while trying to access something from outside of the scope it would error

```
var = "hello"
Proc.new(scope: false) { puts var }.call
# => NameError: undefined local variable or method `var' for main:Object
```

An example of a case where this could be used is found in https://github.com/rails/sprockets/pull/417. In this case we are getting rid of anonymous functions in favor of a method created with refinements. This solution works but it was not an obvious optimization. It would be convenient to have a syntax for defining anonymous functions that do not need access to caller scope.

History

#1 - 11/04/2016 05:30 PM - jeremyevans0 (Jeremy Evans)

It would probably be better if ruby could do analysis and if there are no references to local variables in the outer scope(s), automatically optimize such procs, so that you don't need to specify scope: false. I believe JRuby already does that optimization. I imagine it is fairly difficult to implement though.

In performance sensitive code, I've long used optimizations such as converting:

```
callable = lambda{|arg| puts arg}
```

to:

```
object = Object.new
def object.a(arg) puts arg end
callable = object.method(:a)
```

Having to do this to speed things up is kind of a pain, and I can see where a scope: false option would make manual optimization easier. I'm not against such an option, but if there are any plans to automatically optimize all procs that do not access the surrounding scope, this may not be needed.

#2 - 11/04/2016 06:01 PM - rmosolgo (Robert Mosolgo)

Much +1 for this feature, I used anonymous functions extensively in graphql-ruby until I learned about this overhead. We still have lots of APIs where the user input is a callable, and I would love to be able to advise procs in those cases.

In my case, implementing this feature as an optimization would be great.

I'd be happy to test a patch if/when one is available.

My experience with Ruby source is limited to the C extension API, but if someone can advise where this optimization should be implemented, I'd love to take a look and see if I can learn enough to make it work!

#3 - 11/08/2016 07:15 AM - headius (Charles Nutter)

In https://github.com/rails/sprockets/pull/417 I showed that JRuby can be significantly faster if the proc is stuffed into a define_method method. This is because we optimize define_method blocks to behave exactly like a plain method (with no closure state, etc) if we can statically determine it does not access its captured scope.

This feature could be generalized in any implementation as a function kernel method. When creating a function a Ruby impl would check if any captured state was accessed. This includes variables like $~. If that state is accessed, it could either be a warning or an error. If it is not accessed, then the closure body would be "unrooted" from its captured scope and turned into a flat method. Much of the overhead shown here would go away.

#4 - 11/08/2016 07:18 AM - headius (Charles Nutter)

I did not make clear what I was proposing. I do like the keyword idea, but I think I'd prefer a specific method like function:

```
function { ... }
```

Like define_method in JRuby, this would be an indicator that we explicitly want an unrooted body of code that acts like a pure function without any captured, mutable state.

#5 - 11/18/2016 07:00 PM - justcolin (Colin Fulton)

This would be a fantastic feature. I second the idea of something like function { }, func { }, <> { }, --> { }, ->> { }, or <-> { }.

Would these behave like regular Procs or lambda Procs when it comes to return semantics and arty checking? I personally think it would be useful to have both types, but having four variants of anonymous functions in Ruby might be a bit overwrought.

#6 - 11/19/2016 12:09 AM - phluid61 (Matthew Kerwin)

On 19/11/2016, justcolin@gmail.comjustcolin@gmail.com wrote:

> Issue #12901 has been updated by Colin Fulton.
>
> This would be a fantastic feature. I second the idea of something like
> function { }, func { }, <> { }, --> { }, ->> { }, or <-> { }.
>
> Would these behave like regular Procs or lambda Procs when it comes to
> return semantics and arty checking? I personally think it would be useful to
> have both types, but having four variants of anonymous functions in Ruby
> might be a bit overwrought.

I think it makes sense and would be useful for a 'function' to have
strict arity checking and return from the anonymous function, like a
lambda or a method.

--
Matthew Kerwin
http://matthew.kerwin.net.au/

#7 - 11/22/2016 10:21 PM - rmosolgo (Robert Mosolgo)

ko1 mentioned a similar possibility in his RubyConf talk about Guilds. An "isolated proc" would be a good option to pass between guilds (since it doesn't depend on its binding).

https://youtu.be/mjzmUUQWqco?t=34m26s

#8 - 12/01/2016 03:55 AM - justcolin (Colin Fulton)

It would also be very useful for making ObjectSpace#define_finalizer easier to use (to avoid the common memory leak passing a regular proc in can cause).

#9 - 01/19/2017 07:03 AM - matz (Yukihiro Matsumoto)

The original idea changes the semantics of the language by keyword argument. I don't think that's a good idea.
Some proposed new syntaxes like function, or other forms of stabby lambda. I don't like them either because this only affects the performance.

Koichi has a different idea, and he will submit the proposal.

Matz.

#10 - 01/25/2017 03:13 AM - jwmittag (Jörg W Mittag)

Jeremy Evans wrote:

> It would probably be better if ruby could do analysis and if there are no references to local variables in the outer scope(s), automatically optimize such procs, so that you don't need to specify scope: false. I believe JRuby already does that optimization. I imagine it is fairly difficult to implement though.

"difficult" == "Solving the Halting Problem", unfortunately.

#11 - 01/25/2017 03:28 AM - jwmittag (Jörg W Mittag)

I have thought about this a number of times, but never got around to writing a feature request: we already have a way to tell Ruby that we don't want to capture certain variables from the surrounding scope: block local variables. We also have "wildcard" syntax for block parameters. So, let's just combine the two, to tell Ruby that we don't want to capture any variables:

```
Proc.new {|a, b:, *c; d, e, *| }
# or
-> (a, b:, *c; d, e, *) {}
```

This will make d and e block local variables, and make the block not close over the surrounding environment.

This is a conservative extension of existing syntax.

We can even further extend it to provide a "whitelist" of variables to capture:

```
Proc.new {|a, b:, *c; d, e, *; f, g| }
# or
-> (a, b:, *c; d, e, *; f, g) {}
```

This will make it an error if there are no local variables named f and g in the surrounding scope. The *current* behavior would then be equivalent to:

```
Proc.new {|a, b:, *c;; *| }
# or
-> (a, b:, *c;; *) {}
```

I.e.: no block local variables, capture all variables.

We will need to make a decision, though, as to whether this should include lexical binding of self or not. We could easily allow both explicit opt-out and explicit opt-in by allowing self as a legal identifier in the parameter list. E.g. a block that captures no variables, but *does* capture self would be declared like:

```
Proc.new {|a, b:, *c;* ; self| }
# or
-> (a, b:, *c; *; self) {}
```

A block that captures all variables but does not capture self:

```
Proc.new {|a, b:, *c;self ; *| }
# or
-> (a, b:, *c; self; *) {}
```

and a block that captures *nothing*:

```
Proc.new {|a, b:, *c;*, self| }
# or
-> (a, b:, *c; *, self) {}
```

An open question is: what should the default for capturing self be, when there is only a wildcard, i.e. these two cases:

```
Proc.new {|a, b:, *c;; *| }
# or
-> (a, b:, *c;; *) {}

Proc.new {|a, b:, *c; *| }
# or
-> (a, b:, *c; *) {}
```

I have no preference either way.

#12 - 02/27/2017 04:18 PM - sigsys (Math Ieu)

Would be better if this could be determined automatically, as others have pointed out.

A lot of functional programming languages optimize this away with some form of "lambda lifting".

The shared variables that are not ever reassigned can all be copied into a flat array (no matter their depth) stored with the closure object. This makes it independent from the stack structure. And most closures will not share access to a lot of variables, so this is usually pretty fast (and everything the closure needs can be stashed in a single memory allocation).

Variables that get reassigned need to still be shared though. Some implementations just add an indirection level to those variables (often called "boxes"). This adds extra memory allocations though. This is fine when variables are rarely reassigned (as is often the case in functional languages) or that those at least aren't being shared.

IIRC the LUA C interpreter does something fancy. There's a linking between the closure objects and the stack structure which is undone as the stack unwinds. When a function returns, it will no longer modify its shared variables, so they no longer need to be shared with it (they still need to be shared in-between multiple closure objects created from this scope though).

This all can be determined statically even with dynamic languages as long as their lexical scoping rules are strict enough. The introspection features like the "binding" method and maybe the way "eval" works might make this hard to optimize though (since they essentially capture everything?). So maybe it would have to only be done on-demand (maybe enabled with a magic comment?) to not break things.

#13 - 02/28/2017 01:09 AM - shyouhei (Shyouhei Urabe)

Math Ieu wrote:

> The introspection features like the "binding" method and maybe the way "eval" works might make this hard to optimize though (since they essentially capture everything?).

Yes. This is why we capture everything. And, you have to notice that binding and even evel itself are implemented as normal methods; they are subject to be aliased. So you can't say if a Proc's binding is called or not until it is actually called (recap: ruby's method resolution is dynamic). Never until you actually obtain its binding, a Proc's binding cannot be optimized because of possibility of being obtained later == you can't optimize at all. This is where we stand.

I think it's impossible to optimize lambdas without breaking backwards compatibility.

#14 - 02/28/2017 02:09 AM - rmosolgo (Robert Mosolgo)

What about checking at the block level? I mean, could we check that:

- All local variable reads come from the block's local table
- There are no method calls on implicit receiver

So, this could be optimized:

```
lambda { |a, b| a + b }
```

But this could not be optimized:

```
b = 1
lambda { |a| a + b }
```

Nor could this:

```
lambda { |a, b| c(a + b) }
```

Is it possible to make that kind of check?

#15 - 02/28/2017 04:08 AM - shyouhei (Shyouhei Urabe)

Robert Mosolgo wrote:

> What about checking at the block level? I mean, could we check that:
>
> - All local variable reads come from the block's local table
> - There are no method calls on implicit receiver
>
> So, this could be optimized:
>
> ```
> lambda { |a, b| a + b }
> ```

Sorry, no. You can look at outside scope from within the lambda's binding like this:

```
zsh % rbenv exec irb
irb(main):001:0> a = 1
=> 1
irb(main):002:0> b = lambda {|c, d| c + d }.binding
=> #<Binding:0x007fa672001160>
irb(main):003:0> b.local_variable_set(:a, 2)
```

```
=> 2
irb(main):004:0> a
=> 2
irb(main):005:0>
```

The lambda does not use variable a. But still, you can see through the binding.

#16 - 02/28/2017 10:30 AM - duerst (Martin Dürst)

Robert Mosolgo wrote:

> What about checking at the block level? I mean, could we check that:
>
> - All local variable reads come from the block's local table
> - There are no method calls on implicit receiver
>
> So, this could be optimized:
>
> ```
> lambda { |a, b| a + b }
> ```

In addition to what Shyouhei said, somebody could also redefine addition, like so:

```
alias :old_plus :'+'
def + (other)
  binding # change to something more interesting
  self.old_plus other
end
```

#17 - 02/28/2017 12:35 PM - rmosolgo (Robert Mosolgo)

I didn't think about (er, *know* about!) those ways of accessing the inner scope. That definitely rules out skipping the binding in those cases! Thanks for the info.

#18 - 02/28/2017 08:47 PM - Eregon (Benoit Daloze)

Shyouhei Urabe wrote:

> I think it's impossible to optimize lambdas without breaking backwards compatibility.

It is possible with deoptimization though.
An optimizer might assume that a block does not call binding and in case it does, deoptimize.
Removing the scope lookup overhead for accessing captured variables
is possible if one considers a whole method containing the lambda for analysis.
However, it seems much harder to optimize the scope lookup for a lambda capturing a non-optimized scope.

I do not think scope lookup is the only overhead of lambdas though,
there is a lot of implicitly captured state which needs to be restored for every call
(but the same optimizations apply when optimizing a method containing the lambda).

#19 - 02/28/2017 10:37 PM - sigsys (Math Ieu)

Shyouhei Urabe wrote:

> Math Ieu wrote:
>
> > The introspection features like the "binding" method and maybe the way "eval" works might make this hard to optimize though (since they essentially capture everything?).
>
> Yes. This is why we capture everything. And, you have to notice that binding and even evel itself are implemented as normal methods; they are subject to be aliased. So you can't say if a Proc's binding is called or not until it is actually called (recap: ruby's method resolution is dynamic). Never until you actually obtain its binding, a Proc's binding cannot be optimized because of possibility of being obtained later == you can't optimize at all. This is where we stand.
>
> I think it's impossible to optimize lambdas without breaking backwards compatibility.

But it could be enabled on-demand. Say with a magic comment, then all closures created from code of this file would be optimized to not capture everything. I guess there might as well be special syntax for it then though. Or maybe include something in the module with lambda/proc variants that don't capture everything.

Dunno if "lazy" capture could be a useful compromise? If binding/eval is ever called by/on a closure, then the rest of the variables from the stack is captured. It would only work if it's called before the corresponding levels of the stack unwind though (but full capture could be triggered beforehand if needed).

#20 - 03/01/2017 01:42 AM - shyouhei (Shyouhei Urabe)

Benoit Daloze wrote:

> Shyouhei Urabe wrote:
>
> > I think it's impossible to optimize lambdas without breaking backwards compatibility.
>
> It is possible with deoptimization though.
> An optimizer might assume that a block does not call binding and in case it does, deoptimize.

Let me ask a (possibly stupid) question. Consider this scenario:

1. create a proc.
2. it gets optimized.
3. lots of calculations goes on without variable captures.
4. obtain a binding from it (via deoptimization).

now, should what be visible from the obtained binding? Should we re-calculate the optimized-out variables?

Of course it could be anything if we are allowed to introduce incompatibility but...

> Removing the scope lookup overhead for accessing captured variables
> is possible if one considers a whole method containing the lambda for analysis.
> However, it seems much harder to optimize the scope lookup for a lambda capturing a non-optimized scope.
>
> I do not think scope lookup is the only overhead of lambdas though,
> there is a lot of implicitly captured state which needs to be restored for every call
> (but the same optimizations apply when optimizing a method containing the lambda).

I agree with those parts.

#21 - 03/01/2017 01:56 AM - shyouhei (Shyouhei Urabe)

Math Ieu wrote:

> Shyouhei Urabe wrote:
>
> > Math Ieu wrote:
> >
> > > The introspection features like the "binding" method and maybe the way "eval" works might make this hard to optimize though (since they essentially capture everything?).
> >
> > Yes. This is why we capture everything. And, you have to notice that binding and even evel itself are implemented as normal methods; they are subject to be aliased. So you can't say if a Proc's binding is called or not until it is actually called (recap: ruby's method resolution is dynamic). Never until you actually obtain its binding, a Proc's binding cannot be optimized because of possibility of being obtained later == you can't optimize at all. This is where we stand.
> >
> > I think it's impossible to optimize lambdas without breaking backwards compatibility.
>
> But it could be enabled on-demand. Say with a magic comment, then all closures created from code of this file would be optimized to not capture everything. I guess there might as well be special syntax for it then though. Or maybe include something in the module with lambda/proc variants that don't capture everything.
>
> Dunno if "lazy" capture could be a useful compromise? If binding/eval is ever called by/on a closure, then the rest of the variables from the stack is captured. It would only work if it's called before the corresponding levels of the stack unwind though (but full capture could be triggered beforehand if needed).

I (broadly) agree with you. Extending or modifying the language to make optimization possible is a great idea. Matz said in comment #9 that Koichi has one of such ideas. Let us see what he proposes.

#22 - 03/01/2017 11:11 AM - Eregon (Benoit Daloze)

Shyouhei Urabe wrote:

> Let me ask a (possibly stupid) question. Consider this scenario:

> > 1. create a proc.
> > 2. it gets optimized.
> > 3. lots of calculations goes on without variable captures.
> > 4. obtain a binding from it (via deoptimization).
> >
> > now, should what be visible from the obtained binding? Should we re-calculate the optimized-out variables?

I assume you mean a binding via Kernel#binding.
I think that's the general contract of deoptimization: it must not be observable from the user.
So at the deoptimization point, the current computations must be stored in a real frame,
or be recomputed if they are side-effect free and the deoptimization implements that.

If it is a Proc#binding, then it only captures variables defined outside the Proc,
so variables only defined inside the Proc are not visible to that.
With the technique I mentioned, writes to captured variables can
only be optimized out if the surrounding scope is compiled as well.

#23 - 03/02/2017 01:53 AM - shyouhei (Shyouhei Urabe)

Benoit Daloze wrote:

> > now, should what be visible from the obtained binding? Should we re-calculate the optimized-out variables?
>
> I assume you mean a binding via Kernel#binding.
> I think that's the general contract of deoptimization: it must not be observable from the user.
> So at the deoptimization point, the current computations must be stored in a real frame,
> or be recomputed if they are side-effect free and the deoptimization implements that.

This (recomputation) sounds practically very difficult to me, if not impossible. Much easier to avoid optimization at all when binding is used.

> If it is a Proc#binding, then it only captures variables defined outside the Proc,
> so variables only defined inside the Proc are not visible to that.
> With the technique I mentioned, writes to captured variables can
> only be optimized out if the surrounding scope is compiled as well.

OK, I see. I confused Kernel#binding and Proc#binding. Thank you.

#24 - 03/02/2017 03:08 AM - ko1 (Koichi Sasada)

On 2017/03/01 5:47, eregontp@gmail.com wrote:

> It is possible with deoptimization though.
> An optimizer might assume that a block does not call binding and in case it does, deoptimize.

I haven't read discussion details, but some optimizations eliminate
frames (local variables) and deoptimization can not recover eliminated
variables. I don't think most of people needs such strict compatibility,
but I don't think "possible" is not exact.

My rough idea is forcing calling "binding" to people who want to use
local variables outside from this block. It breaks compatibility and
"binding" will be a magical sentence (like pragma, not Ruby-like way).
But not so bad.

--
// SASADA Koichi at atdot dot net

#25 - 03/02/2017 10:30 AM - Eregon (Benoit Daloze)

Koichi Sasada wrote:

> I haven't read discussion details, but some optimizations eliminate
> frames (local variables) and deoptimization can not recover eliminated
> variables.

No, that's the important part, proper deoptimization *must* be able to restore
local variables. It can do so how it wants, but typically that involves
having some kind of mapping from registers and stack locations to
frame variables. During deoptimization, these values from registers

and the stack are written in the frame, and it's exactly the same as if
the interpreter ran the code before that.

Shyouhei Urabe wrote:

> This (recomputation) sounds practically very difficult to me, if not impossible. Much easier to avoid optimization at all when binding is used.

Indeed, I think it's only done for very simple things like arithmetic operations,
where the value can restored from operands and recomputation has no observable side-effect.

But it's impossible to know if Kernel#binding can be used or not (due to aliasing).

#26 - 03/02/2017 10:44 AM - Eregon (Benoit Daloze)

To come back to the main topic of this issue,
I think having some syntax support for a non-capturing lambda
can be a useful feature in Ruby, but should not be for performance reasons.
It should be a design tool.

If a lambda/proc does not use captured variables, there is no reason
for it to lookup parent scopes. So there should not be any scope lookup overhead
if a lambda/proc does not capture variables.

What is the source of the overhead for calling lambdas on MRI?
Is it restoring the implicit state (self, cref, a pointer to the parent frame
in case #binding is called, etc) that must be restored? Some extra checks?

Kernel#binding applies to all kinds of lambdas/procs and
that affects how frames can be represented but it is mostly orthogonal
to this issue (methods also have to care about Kernel#binding).

#27 - 03/03/2017 08:40 AM - shyouhei (Shyouhei Urabe)

Benoit Daloze wrote:

> Shyouhei Urabe wrote:
>
> > This (recomputation) sounds practically very difficult to me, if not impossible. Much easier to avoid optimization at all when binding is used.
>
> Indeed, I think it's only done for very simple things like arithmetic operations,
> where the value can restored from operands and recomputation has no observable side-effect.
>
> But it's impossible to know if Kernel#binding can be used or not (due to aliasing).

True. There are two ways to tackle this problem:

1. add a new construct that does not capture variables outside of a given block.
2. just prohibit obtaining bindings; make lambdas default optimizable.
    - optionally with something that is statically analyzable to "mark" a lambda to be "volatile", and allow obtaining bindings only within that block.

I understand Richard's proposal is #1 while Koichi suggests #2. #2's advantage is that existing codes can benefit. #1 on the other hand won't break existing codes which might be a good property. It seems a matter of allow or disallow binding by default.

#28 - 05/27/2020 05:20 PM - dsisnero (Dominic Sisneros)

Can we visit this again which if we go with 1:) add a new construct we could solve https://bugs.ruby-lang.org/issues/11630 - serializing functions
also, if we add new constructs we can add a multi-method version - https://bugs.ruby-lang.org/issues/16254

func sum(a,b){ a + b}

funcm foo(a){ Primitive.foo1(a)}

funcm foo(a,b){ Primitive.foo2(a,b)}

#29 - 06/14/2020 11:48 AM - Eregon (Benoit Daloze)

dsisnero (Dominic Sisneros) What's the self in those "anonymous functions"?
I think no self would be very confusing in Ruby, e.g., puts some_expression wouldn't work.

Serializing a Proc is hard because it needs to serialize code and captured objects.
If you need to serialize self, I think it's no easier than serializing a regular Proc
(which is theoretically possible, but hard, and e.g., what if the process deserializing doesn't have the class of self?).

BTW, they're not anonymous in your case, so it seems you're conflating many issues in one here.

multi-method also seems orthogonal to me.

#30 - 06/16/2020 08:11 AM - ko1 (Koichi Sasada)

I read this ticket and several comments:

Proc#call calling performance

The performance gap is because of optimization level. We didn't optimize Proc#call enough, so the capturing is not a reason.

capturing outer-scope (making closure)

We will introduce Proc#isolate which creates a Proc but accessing outer scope is not allowed for the parallel execution (Guild/Ractor). So we can make such Proc. However, the normal Proc is created before calling Proc#isolate (Proc.new{...}.isolate), So Proc capturing overhead is not avoided. Proc.new(isolate: true){ ... } can avoid it. New syntax for -> is also considerable (but not sure it is valuable).

BTW self is same as outer-scope. Guild/Ractor replaces it using instance_eval technique.

#31 - 06/17/2020 04:45 PM - dsisnero (Dominic Sisneros)

I want to be able to easily make a proc like object that is not a closure. I want to avoid capturing overhead - hence a separate syntax. Because we are not capturing the closure- I was hoping to be able to serialize the object. Basically Proc.new(isolate: true) with a new syntax to avoid capturing overhead

#32 - 06/17/2020 11:41 PM - nobu (Nobuyoshi Nakada)

If it doesn't capture the receiver, it looks similar to UnboundMethod than Proc.

How about this syntax:

```
plus = def->(a) = self + a
plus.bind_call(1, 2) #=> 3
plus_1 = plus.bind(1)
plus_1.call(11) #=> 12
```