Recently I discovered that one could use Kernel#binding to capture the environment of a frame that is not directly below the stack frame for Kernel#binding. I've known that C extensions have this privilege for a while, but didn't realize that it was also possible using only the core library. This is a powerful primitive that allows for some funky programs:

```ruby
def lookup(hash, key)
  hash[key]
end
```

```ruby
p lookup(Hash.new{ Kernel.instance_method(:send).method(:bind_call).to_proc >>
  ->(binding) { binding.local_variable_set(':hash, :action_at_a_distance!') } })
```

There might be ways to compose core library procs such that it's less contrived and more useful, but I couldn't figure out a way to do it. Maybe there is a way to make a "local variable set" proc that takes only a name-value pair and no block?

**What's the big deal?**

This behavior makes the implementation language of methods part of the API surface for Kernel#binding. In other words, merely changing a Ruby method to be a C method can be a breaking change for the purposes of Kernel#binding, even if the method behaves the same in all other observable ways. I feel that whether a method is native or not should be an implementation detail and should not impact Kernel#binding.

This is a problem for Ruby implementations that want to implement many core methods in Ruby, because they risk breaking compatibility with CRuby. TruffleRuby has this problem as I alluded to earlier, and CRuby risks making unintentional breaking changes as more methods change to become Ruby methods in the core library.

**Leaking less details**

I think a straight forward way to fix this issue is by making it so that Kernel#binding only ever looks at the stack frame directly below it. If the frame below is a not a Ruby frame, it can return an empty binding. I haven't done the leg work of figuring out how hard this would be to implement in CRuby, though. This new behavior allows observing the identity of native frames, which is new.

**Does the more restrictive behavior help YJIT?**
Maybe. It's hard to tell without building out more optimizations that are related to local variables. YJIT currently doesn't do much in that area. If I had to guess I would say the more restrictive semantics at least open up the possibility of some deoptimization strategies that are more memory efficient.

**What do you think?**

This is not a huge issue, but it might be nice to start thinking about for the next release. If a lot of people actually rely on the current behavior we can provide a migration plan. Since it might take years to land, I would like to solicit feedback now.

---

**History**

#1 - 01/13/2022 11:38 PM - alanwu (Alan Wu)
- Description updated

#2 - 01/14/2022 07:10 AM - mame (Yusuke Endoh)

> Interesting. I created a simpler version.

```ruby
class Magic
  define_singleton_method :modify_caller_env!, method(:binding).to_proc >->(bindg) { bindg.local_variable_set(:my_var, 42) }
end

my_var = 1
Magic.modify_caller_env!
p my_var #=> 42
```

I have no strong opinion, but your solution "Kernel#binding only ever looks at the stack frame directly below it" looks reasonable to me.

BTW, as you may know, there is a relatively popular gem called [binding_of_caller](https://github.com/ruby/ruby/ruby/blob/master/standardlib/binding.rb) to extract a Binding object from caller frames. YJIT optimization might be still difficult even after Kernel#binding was changed.

#3 - 01/22/2022 03:13 AM - jeremyevans0 (Jeremy Evans)

I submitted a pull request to make Kernel#binding only look up a single frame, which fixes the issue: [https://github.com/ruby/ruby/pull/5476](https://github.com/ruby/ruby/pull/5476). Not sure if all the semantics in the pull request are desired (i.e. eval and receiver raise `RuntimeError` for bindings for non-Ruby frames), so this is probably worth discussing at the next developer meeting.

#4 - 01/22/2022 04:05 PM - Eregon (Benoit Daloze)

Nice find!
Agreed this should be fixed, and Kernel#binding should never provide access to anything but its direct caller method's frame (whether that's defined in Ruby, C or anything).
In other words Kernel#binding should provide access to the local variables immediately around the call to Kernel#binding.
The current behavior in CRuby is effectively breaking encapsulation, even though I'd think it never intended that.

#5 - 02/17/2022 01:17 AM - alanwu (Alan Wu)

To simplify the semantics and implementation, we could make Kernel#binding raise when the direct caller is not Ruby. I think it's reasonable given that the Binding class was designed for Ruby and doesn't necessarily make sense for other languages.

#6 - 02/17/2022 07:55 AM - matz (Yukihiro Matsumoto)

Okay, binding should raise an exception when called from a C defined method.

Matz.

#7 - 02/17/2022 01:04 PM - Eregon (Benoit Daloze)

FWIW TruffleRuby currently raises one of these 2 errors when trying to call a Ruby method which needs a direct Ruby frame above:

```ruby
Cannot call Ruby method which needs a Ruby caller frame directly in a foreign language (RuntimeError) or Foo#bar needs the caller frame but it was not passed (cannot be called directly from a foreign language) (RuntimeError)
```

03/16/2022
That can happen for C extension but also for any other language calling Ruby methods (e.g. JS/Python/etc).

#8 - 02/17/2022 10:52 PM - jeremyevans0 (Jeremy Evans)

matz (Yukihiro Matsumoto) wrote in #note-6:

Okay, binding should raise an exception when called from a C defined method.

I've submitted a pull request for this: https://github.com/ruby/ruby/pull/5567

It's trickier than I expected, and took some trial and error to get right. I also found that some tests were implicitly relying on the previous behavior. One case was related to tracing, as set_trace_func yields bindings. I modified the logic there so that cases where generating the binding would raise an exception, we yield nil as the binding (this was already done in some cases, so I don't think there should be significant backwards compatibility issues).