I'm extremely surprised (and disappointed) that, currently:

```
x = nil
x.&.foo.&.bar #=> NoMethodError: undefined method `bar' for nil:NilClass
```

To make it safe, you have to write `x.&.foo.&.bar`. But if `foo` is never supposed to return `nil`, then that code isn't "fail early" in case it actually does. `nil.&.foo.&.bar` is more expressive, simpler and is perfect if you want to an error if `foo` returned `nil`. To actually get what you want, you have to resort using the old form `x && x.foo.bar`...

In CoffeeScript, you can write `x()?.foo.bar` and it will work well, since it gets compiled to

```
if ((_ref = x()) != null) {
  _ref.foo.bar;
}
```

All the discussion in [#11537](https://github.com/ruby/ruby/issues/11537) focuses on `x.&.foo.&.bar`, so I have to ask:

Matz, what is your understanding of `x.&.foo.bar`?

I feel the current implementation is not useful and should be changed to what I had in mind. I can't see any legitimate use of `x.&.foo.bar` currently.
compile.c: fix dangling link

- compile.c (iseq_peephole_optimize): should not replace the current target INSN, not to follow the replaced dangling link in the caller. [ruby-core:74993] [Bug #11816]

Revision 4989a071 - 04/18/2016 08:53 AM - naruse (Yui NARUSE)
merge revision(s) 54628: [Backport #12296]

 compile.c (iseq_peephole_optimize): should not replace the current target INSN, not to follow the replaced dangling link in the caller. [ruby-core:74993] [Bug #11816]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/branches/ruby_2_3@54635 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 54635 - 04/18/2016 08:53 AM - naruse (Yui NARUSE)
merge revision(s) 54628: [Backport #12296]

* compile.c (iseq_peephole_optimize): should not replace the current target INSN, not to follow the replaced dangling link in the caller. [ruby-core:74993] [Bug #11816]

History

#1 - 12/14/2015 06:30 PM - marcandre (Marc-Andre Lafortune)
BTW, this came up in Matt Larraz's PR https://github.com/ruby/ruby/pull/1142
Both Hiroshi Shibata and Benoit Daloze thought (as I did) that e_error_bytes&.dup.force_encoding(...) should work.

#2 - 12/14/2015 11:00 PM - nobu (Nobuyoshi Nakada)
I haven't thought it at all, but it seems interesting.

#3 - 12/14/2015 11:47 PM - mame (Yusuke Endoh)
It seems very sensitive to determine how long it propagates.

How should we interpret str&.upcase + "foo"?

If it is considered as str&.upcase&.+("foo"), it may return nil (as I expect).
If it is considered as (str&.upcase) + "foo", it may raise an exception. We need write p str&.upcase&.+("foo").

How's that for str&.upcase == nil?

If it is considered as str&.upcase&.==(nil), it will return false or nil. Very confusing.
If it is considered as (str&.upcase) == nil, it will return true or false.

--
Yusuke Endoh mame@ruby-lang.org

#4 - 12/15/2015 03:49 AM - nobu (Nobuyoshi Nakada)
If it propagates, we could write safe aref as:

ary&.itself[idx]
ary&.itself[idx] = value

#5 - 12/15/2015 04:25 AM - matz (Yukihiro Matsumoto)
Marc,

We need more clarification. Did you mean all method calling dots should be interpreted as &.?
What about other calls like array/hash indexing or operators?

Matz.

#6 - 12/15/2015 06:01 PM - marcandre (Marc-Andre Lafortune)
Glad to see we're headed towards having x.&.foo.bar not be the same as (x.&.foo).bar.

When we ask about &. vs operators, we can see that the question is similar to that of precedence, right? We could say that currently in trunk, &. has same "precedence" as ., which we now agree is not ideal. We need to bump &. down at least one level.
So either &. has a "precedence" just lower than ., or we (well, Matz) decide to make it even lower.

My guess is that the ideal would if &. has a "precedence" in between comparison operators (<, =>, ...) and equality operators (==, <=>, ...)

This way:

```
x = nil
x&.foo.bar #=> nil
x&.[42] #=> nil
x&.[42] = 43 #=> nil
x&.foo * 42 #=> nil
x&.foo + 42 #=> nil
x&.foo << 42 #=> nil
x&.foo < 42 #=> nil
x&.foo == 42 #=> false  ### This is where the precedence of &. is higher
x&.foo | 42 #=> 42
x&.foo ? 1 : 2 #=> 2
```

My reasoning is that this divides the operators cleanly into those that we never want to apply to nil (e.g. +), from those that can be applied to it (e.g. ||).

Did you mean all method calling dots should be interpreted as &.?

Just to be clear, no, there should be a difference between x&.foo&.bar and x&.foo.bar. The following should be equivalent:

```
x&.foo.bar
# same as
if temp = x&.foo
  temp.bar
end
```

In particular:

```
x = Object.new
def x.foo; nil; end
x&.foos.bar #=> nil
x&.foo.bar #=> NoMethodError, no method :bar for nil
```

My naïve understanding is that foo&.bar should be a shorthand for foo && foo.bar, and therefore the &. operator should take the same level of precedence as the && operator.

Marc-Andre Lafortune wrote:

```
x = nil
x&.foo.bar #=> nil
x&.[42] #=> nil
x&.[42] = 43 #=> nil
x&.foo * 42 #=> nil
x&.foo + 42 #=> nil
x&.foo << 42 #=> nil
x&.foo < 42 #=> nil
x&.foo == 42 #=> false  ### This is where the precedence of &. is higher
x&.foo | 42 #=> 42
x&.foo ? 1 : 2 #=> 2
```

If you substitute x&.foo for x && x.foo in the above, they all evaluate the same except for one:

```
x && x.foo #=> 42 #=> nil
```

This makes sense, since && is a method and not an operator.

Matz, did you get a chance to think about the "precedence" level of &.? It would be harder to change after christmas...

If you substitute x&.foo for x && x.foo in the above, they all evaluate the same except for one:

```
x && x.foo #=> 42 #=> nil
```

This makes sense, since && is a method and not an operator.

Matz, did you get a chance to think about the "precedence" level of &.? It would be harder to change after christmas...

If you substitute x&.foo for x && x.foo in the above, they all evaluate the same except for one:

```
x && x.foo #=> 42 #=> nil
```

This makes sense, since && is a method and not an operator.

Matz, did you get a chance to think about the "precedence" level of &.? It would be harder to change after christmas...

If you substitute x&.foo for x && x.foo in the above, they all evaluate the same except for one:

```
x && x.foo #=> 42 #=> nil
```

This makes sense, since && is a method and not an operator.
Matz, did you get a chance to think about the "precedence" level of &.? It would be harder to change after christmas...

Is it really hard to change after the release of Ruby 2.3?

IMHO, the current behavior of x&.foo.bar is useless, so users have to use x&.&.foo&.bar instead. Even if the behavior of x&.foo.bar is changed as you expect, x&.&.foo&.bar still works, and users can switch from &.&.foo&.bar to x&.foo.bar gradually.

Please tell me if I miss anything.

I'm not against the proposal itself, but there's no enough time left....

FYI, the behavior of Groovy's ? is useless, so users have to use x&.&.foo&.bar instead.

Even if the behavior of x&.foo.bar is changed as you expect, x&.&.foo&.bar still works, and users can switch from x&.&.foo&.bar to x&.foo.bar gradually.

Please tell me if I miss anything.

I'm not against the proposal itself, but there's no enough time left....

FYI, the behavior of Groovy's ? seems to be the same as Ruby's &.

$ groovy -e 'x = null; print(x?.foo.bar)'
Caught: java.lang.NullPointerException: Cannot get property 'bar' on null object
    at script_from_command_line.run(script_from_command_line:1)

Please tell me if someone knows the behavior in C#.

#11 - 12/17/2015 08:36 AM - marcandre (Marc-Andre Lafortune)

Shugo Maeda wrote:

Is it really hard to change after the release of Ruby 2.3?

True, it could be changed afterwards, although it is far from ideal.

Please tell me if someone knows the behavior in C#.

I don't know, but I can say that both CoffeeScript and Swift do the correct thing with respect to x&.foo.bar.

#12 - 12/17/2015 12:04 PM - mame (Yusuke Endoh)

Shugo Maeda wrote:

Is it really hard to change after the release of Ruby 2.3?

As Marc-Andre stated in https://bugs.ruby-lang.org/issues/11816#note-7, it will bring incompatibility. If we may change the spec in future, I think we should explicitly state the possibility in doc and release message. Also, it might be a good idea to mark the feature as "experimental".

Another idea for the proposal: how about propagating &. as long as explicit method chain (that uses . literally) continues?

x&.foo * 42 == (x&.foo) * 42
x&.foo.*(42) == (x&.foo.*(42)) # strictly not equivalent in the case where #foo returns nil

x&.foo[42] is still ambiguous. `(x&.foo)[42]` looks simple and consistent to me.

--
Yusuke Endoh mame@ruby-lang.org

#13 - 12/17/2015 01:32 PM - shugo (Shugo Maeda)

Yusuke Endoh wrote:

Is it really hard to change after the release of Ruby 2.3?

As Marc-Andre stated in https://bugs.ruby-lang.org/issues/11816#note-7, it will bring incompatibility. If we may change the spec in future, I think we should explicitly state the possibility in doc and release message. Also, it might be a good idea to mark the feature as "experimental".

It would be better if Matz would like to change the behavior in future.

Another idea for the proposal: how about propagating &. as long as explicit method chain (that uses . literally) continues?

x&.foo * 42 == (x&.foo) * 42
x&.foo.*(42) == (x&.foo.*(42)) # strictly not equivalent in the case where #foo returns nil
In my understanding, rather than (x&.foo.*(42)), x&.foo.*(42) is equivalent to x && x.foo.*(42) except when x is false. So, the phrase "propagating &." is confusing, isn't it?

Anyway, your proposal sounds reasonable because x&.foo * 42 is parsed as (x && x.foo) * 42 rather than x && (x.foo * 42) in my brain. However, the behavior like (x && x.foo) * 42 seems to be useless for the same reason as the current behavior of x&.foo.bar, so there is a trade-off.

#14 - 12/18/2015 04:06 AM - shugo (Shugo Maeda)
Shugo Maeda wrote:

Anyway, your proposal sounds reasonable because x&.foo * 42 is parsed as (x && x.foo) * 42 rather than x && (x.foo * 42) in my brain.

Just for clarification, x&.foo.bar as (x && x.foo).bar looks more natural to me than x && x.foo.bar.

However, the behavior like (x && x.foo) * 42 seems to be useless for the same reason as the current behavior of x&.foo.bar, so there is a trade-off.

So there is also a trade-off between the current behavior and Endoh-san's proposal.

#15 - 12/18/2015 04:27 AM - naruse (Yui NARUSE)
I understand this as the idea returned value of safe navigation operator behaves like NaN while the method chain.
It sounds interesting and feels bit useful but I weakly object it.

Anyway using && in &.'s explanation should be harmful especially this serious topic because the behavior is different in handling false.

#16 - 12/20/2015 05:14 AM - marcandre (Marc-Andre Lafortune)
Matz, is it possible to have a decision before the official release, so we at least know what the future might bring and can introduce it properly?

I still hope my "whichever is the most useful" proposal is accepted. We look at it right now with a mental magnifying glass, but I think that it would become second nature in practice, as I feel it would usually be what the rubyist wants. To me, it feels similar to super which passes argument and block; not necessarily intuitive when you think about it, but if you don't think about it, it does usually what you want to do.

Thanks

#17 - 01/20/2016 02:36 AM - marcandre (Marc-Andre Lafortune)
Dear Matz,

Any update on the precedence of &.?

#18 - 01/20/2016 10:01 AM - matz (Yukihiro Matsumoto)
Marc,

I don't feel it's right to ignore nils in calls after &.
Nil-ignorance should be explicit, I think.

Matz.

#19 - 01/20/2016 06:55 PM - marcandre (Marc-Andre Lafortune)
So if I understand correctly, you like the current behavior, although it is not useful in any circumstance I can think of?

e_error_bytes.dup.force_encoding(...) #=> NoMethodError if error_bytes is nil

So there are basically no circumstances where one would write foo&.bar and follow it with ., <, ==, etc

Even Swift allows x&.foo.bar to be something meaningful.

#20 - 02/24/2016 05:52 PM - seanlinsley (Sean Linsley)
I prefer Marc's proposal here: https://bugs.ruby-lang.org/issues/11816#note-6. I think that's the much more natural than the existing behavior. I found this ticket after being surprised by the behavior, as I attempted to update my project to use &.. Most of the places I would want to use &. currently have an if condition wrapping them, because the first couple objects in the chain may not exist. With the feature as it is currently, it's pretty much useless, because this is unacceptably hard to read:

book&.authors&.last&.first_name

#21 - 03/17/2016 06:34 AM - shyouhei (Shyouhei Urabe)
I was not active when this topic was hot, so to know the current status I asked this on this month's developer meeting. The bottom line is there was no concrete answer for this issue yet.

At first "propagating &. as long as explicit method chain continues" seemed to be the right way. But we found there still are edge cases that should be clearly described before adopting that way. Consider:

```
a&.b.c += d
```

here, if it was a.b.c += d, it calls two methods a.b.c and a.b.c=. Then what should happen for &.?

**#22 - 03/17/2016 06:17 PM - marcandre (Marc-Andre Lafoutine)**

I feel that a&.b.c += d is still an assignment, so should be treated with the same precedence.

```
a = nil
a&.b.c += d  # => nil
a&.b.c ||= d  # => nil
a&.b.c &&= d  # => nil
# etc...
```

I think this is the most helpful solution.

Moreover, we already have (in 2.3.0)

```
a&.b += d  # => nil
a&.b ||= d  # => nil
a&.b &&= d  # => nil
```

Given this, I feel the resolution I'm giving is also the most intuitive.

Just to be sure, with "propagating &. as long as explicit method chain continues", we consider that x&.foo * 42 returns nil, as calling " is equivalent to ", right?

**#23 - 03/18/2016 06:18 AM - nobu (Nobuyoshi Nakada)**

Marc-Andre Lafoutine wrote:

> Just to be sure, with "propagating &. as long as explicit method chain continues", we consider that x&.foo * 42 returns nil, as calling " is equivalent to ", right?

I think so.

**#24 - 04/08/2016 11:42 AM - sawa (Tsuyoshi Sawada)**

What about allowing parentheses after the safe navigation operator like this:

```
foo&.(bar.bar)
```

which should behave in the way proposed.

The edge cases:

Shyouhei Urabe wrote:

```
a&.b.c += d
```

with the two possibilities involved can be distinguished as:

```
a&.(b.c) += d  # non-useful one
a&.(b.c += d)
```

Does this conflict with the current syntax?

Or, if bar above is confusing (to human) with a local variable, then another option may be:

```
foo&(.bar.bar)
```

**#25 - 04/08/2016 12:44 PM - nobu (Nobuyoshi Nakada)**

- **Description updated**

Tsuyoshi Sawada wrote:
Does this conflict with the current syntax?

Yes.

```ruby
proc{|x|p x}.(1) #=> 1
```

Or, if bar above is confusing (to human) with a local variable, then another option may be:

```ruby
foo{.bar.baz}
```

Is &. a single token?

---

**#26 - 04/18/2016 12:36 AM - joanbm (Joan Blackmoore)**

I can understand the intention behind the proposal, but would like to express a (strong) disagreement with it.

The thing is that it would break language's consistency and is in contradiction to declared behavior.

"safe navigation operator" serves, by accepted feature request #11537, as an alternate way of method invocation. The only difference to standard method sending is ignoring passed argument if target object is `nil`. This means, expression may result either in a value of expected type or `nil` and programmer need count with this eventuality.

This bug report, or rather feature request, asks for change of existing role of `. as a "safe" methods chain navigator and make it later/last resort objects combinator.

Lowering operator's precedence would result in much harder to understand code, in figuring out at which moment it would be applied and what would be its argument. In comparison to unambiguous situation, when it has the highest implicit precedence, like the standard dot method call.

Setting an explicit precedence by parens would remedy the situation but, as already mentioned by Nobu, it would clash with existing .() syntax, a very unfortunate design decision introduced into the language as a parse-time way of writing Proc/Method#call. Confusing with standard method arguments passing, enforcing eval precedence, optional dot at operator methods… Unless widely used, its removal in Ruby 3.0 would be highly welcomed.

I'd suggest keep the existing behavior or possibly extend with #respond_to? method check and get rid of exception to the rule when applied to NilClass instance, like nil&.nil? resulting in true.

It may also be taken into the consideration a little inconsistency when followed by operator method:

```ruby```
optional dot omitted
40+2 # 42
40&.+2 # 42
```

vs

```ruby```
40.+2 # 42
40&..+2 # syntax error
```

---

**#27 - 04/18/2016 01:11 AM - phluid61 (Matthew Kerwin)**

I don't necessarily disagree with the rest of Joan's post, but for this point:

Joan Blackmoore wrote:

> It may also be taken into the consideration a little inconsistency when followed by operator method:

```ruby```
optional dot omitted
40+2 # 42
40&.+2 # 42
```

vs

```ruby```
40.+2 # 42
40&..+2 # syntax error
```

That's perhaps not the right consistency to be searching for; the &. operator is a dressed up . operator. There's no equivalent sugar-coated safe operator syntax.

```ruby```
40 + 2  # "unsafe"
#???    # "safe"
40+.2   # "unsafe"
40&+.2  # "safe"
```
Thought about it again and would agree with the last paragraph. Direct substitution is not appropriate here, despite it sounds logical. The \&. operator is a strange beast as other general rules also won't apply, like (optional) preceding dot, ie. object\&.hash is also a syntax error.

Btw. by playing with different \&. contained expressions, I've discovered a possible bug:

This runs ok

```ruby
a = nil
a.\&.foo &&= false # nil
```

Following however freezes VM and need to be SIGKILLED

```ruby
nil.\&.foo &&= false
```

Even at bytecode compilation

```ruby
RubyVM::InstructionSequence.compile('nil&.foo ||= 42')
```

tail of strace output

```ruby
lstat("/usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64", {st_mode=S_IFDIR|0755, st_size=139264, ...}) = 0
lstat("/usr/lib64/ruby", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64/ruby/2.3.0", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64/ruby/2.3.0/unicode_normalize.rb", {st_mode=S_IFREG|0644, st_size=3265, ...}) = 0
```

Tested with

```ruby
ruby -v
ruby 2.3.0p75 (2016-04-07 revision 54505) [x86_64-linux]
```

command used

```ruby
strace ruby --disable=gems,rubyopt -e 'nil&.foo &&= false'
```

#28 - 04/18/2016 05:34 AM - joanbm (Joan Blackmoore)

Matthew (Matthew Johnson)

Thought about it again and would agree with the last paragraph. Direct substitution is not appropriate here, despite it sounds logical. The \&. operator is a strange beast as other general rules also won't apply, like (optional) preceding dot, ie. object\&.hash is also a syntax error.

Btw. by playing with different \&. contained expressions, I've discovered a possible bug:

This runs ok

```ruby
a = nil
a.\&.foo &&= false # nil
```

Following however freezes VM and need to be SIGKILLED

```ruby
nil.\&.foo &&= false
```

Even at bytecode compilation

```ruby
RubyVM::InstructionSequence.compile('nil&.foo ||= 42')
```

tail of strace output

```ruby
lstat("/usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64", {st_mode=S_IFDIR|0755, st_size=139264, ...}) = 0
lstat("/usr/lib64/ruby", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64/ruby/2.3.0", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/usr/lib64/ruby/2.3.0/unicode_normalize.rb", {st_mode=S_IFREG|0644, st_size=3265, ...}) = 0
```

Tested with

```ruby
ruby -v
ruby 2.3.0p75 (2016-04-07 revision 54505) [x86_64-linux]
```

command used

```ruby
strace ruby --disable=gems,rubyopt -e 'nil&.foo &&= false'
```

#29 - 04/18/2016 07:07 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset r54628.

compile.c: fix dangling link

- compile.c (iseq_peephole_optimize): should not replace the current target INSN, not to follow the replaced dangling link in the caller. [ruby-core:74993] [Bug #11816]

#30 - 04/18/2016 07:13 AM - nobu (Nobuyoshi Nakada)

- Status changed from Closed to Open

- Backport changed from 2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN to 2.1: DONTNEED, 2.2: DONTNEED, 2.3: UNKNOWN

#31 - 04/18/2016 08:28 AM - naruse (Yui NARUSE)

- Precedes Bug #12296: [Backport] fix dangling link added

#32 - 04/18/2016 08:53 AM - naruse (Yui NARUSE)

- Status changed from Open to Closed

Applied in changeset ruby_2_3r54635.

merge revision(s) 54628: [Backport #12296]

- compile.c (iseq_peephole_optimize): should not replace the current target INSN, not to follow the replaced dangling link in the caller. [ruby-core:74993] [Bug #11816]
Matz, did you get a chance to consider the precedence of &.?

As a reminder, there's currently no real use for foo&.bar.baz or similar. We are forced to write foo&.bar&.baz even though this could introduce unwanted errors, e.g. if bar was erroneously returning nil.

I still believe that my proposal https://bugs.ruby-lang.org/issues/11816#note-6 is the right one.

Thanks for the consideration.

I found real use case in 【アンチパターン】全部nil(null)かもしれない症候群.

if friend.message.blank?

I don't think that is a real use case. The linked article mentions coding style. It suggests not to use the safe navigation operator without thinking or when it is not necessary. The feature proposed in this thread actually makes these things happen automatically, and will remove the necessity of such consideration. It is actually consistent with the current proposal.

I think .nil?, .inspect, .tap could all be legitimately used on a value or nil.

It makes sense to me that &. is just a safe version of ., not some sort of "infectious nil" operation that propagates until an arbitrary precedence limit is hit. If there was a way to explicitly signal the end of the "infectious nil" I'd probably find it useful (for example, a nicer version of x&.instance_eval{foo.bar}), but then I don't think that's &.; it's something new.

If there was a way to explicitly signal the end of the "infectious nil" I'd probably find it useful
There is one way, and it is the same as with all the cases where the precedence doesn't go the way you want it: parentheses.

(foo || bar) && baz
(friend&.message).blank?

The whole point of precedence is to allow writing things simply and without parentheses most of the time.

Matz: was this discussed at the developers’ meeting?

#40 - 12/06/2017 04:06 AM - phluid61 (Matthew Kerwin)

marcandre (Marc-Andre Lafortune) wrote:

    phluid61 (Matthew Kerwin) wrote:

        If there was a way to explicitly signal the end of the "infectious nil" I'd probably find it useful

    There is one way, and it is the same as with all the cases where the precedence doesn't go the way you want it: parentheses.

        (foo || bar) && baz
        (friend&.message).blank?

    The whole point of precedence is to allow writing things simply and without parentheses most of the time.

Hmm, that's almost right, but in my mind it still doesn't quite fit. I think the problem I have is the expectation that modifying one message dispatch (. → &.) shouldn't affect subsequent messages. If you want to affect the dispatch of a group of messages you should use a scoping construct, and operator precedence (even with parens) isn't scope. That's why foo&.instance_eval{bar.baz} feels right, even if it's ugly.

Perhaps I am alone in seeing &. as an armoured ., not an executing &&. (Despite the fact that false&.! == true)