Constant declaration does not conform to JIS 3017:2013

The order of evaluation in constant declaration does not conform to JIS 3017:2013 11.4.2.2.3.

Problem

Suppose that we are evaluating the following program.

```ruby
eexpr::C = rhs
```

The standard seems to be requiring the following evaluation order:

1. `expr`
   - raise a `TypeError` if the value is not a kind of `Module`
2. `rhs`
3. `rb_const_set(expr, :C, rhs)`

However, the actual implementation evaluates in the following order:

1. `rhs`
2. `expr`
3. `rb_const_set(expr, :C, lhs)`
   - raise a `TypeError` if the `expr` is not a kind of `Module`

How to reproduce

The next program does not raise "recv" but raises "value"

```ruby
raise("recv")::C = raise("value")
```

The next program does not raise a `TypeError` but raises a `RuntimeError`

```ruby
A = 1
A::C = raise("value")
```

Question

- Is this interpretation of the standard correct?
- If it is, Should we change the current behavior?
- If we shouldn't, does it mean an issue in the standard?

c.f.

- [https://twitter.com/n0kada/status/1140234416175763456](https://twitter.com/n0kada/status/1140234416175763456)

Related issues:

Related to Ruby master - Bug #4443: odd evaluation order in a multiple assign... Closed
Fix constant assignment evaluation order

Previously, the right hand side was always evaluated before the left hand side for constant assignments. For the following:

```
lhs::C = rhs
```

rhs was evaluated before lhs, which is inconsistent with attribute assignment (lhs.m = rhs), and apparently also does not conform to JIS 3017:2013 11.4.2.2.3.

Fix this by changing evaluation order. Previously, the above compiled to:

```
0000 putself                                                          (   1)[Li]
0001 opt_send_without_block                 <calldata!mid:rhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0003 dup
0004 putself
0005 opt_send_without_block                 <calldata!mid:lhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0007 setconstant                            :C
0009 leave
```

After this change:

```
0000 putself                                                          (   1)[Li]
0001 opt_send_without_block                 <calldata!mid:lhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0003 putself
0004 opt_send_without_block                 <calldata!mid:rhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0006 swap
0007 topn                                   1
0009 swap
0010 setconstant                            :C
0012 leave
```

Note that if expr is not a module/class, then a TypeError is not raised until after the evaluation of rhs. This is because that error is raised by setconstant. If we wanted to raise TypeError before evaluation of rhs, we would have to add a VM instruction for calling vm_check_if_namespace.

Changing assignment order for single assignments caused problems in the multiple assignment code, revealing that the issue also affected multiple assignment. Fix the multiple assignment code so left-to-right evaluation also works for constant assignments.

Do some refactoring of the multiple assignment code to reduce duplication after adding support for constants. Rename struct masgn_atrasgn to masgn_lhs_node, since it now handles both constants and attributes. Add add_masgn_lhs_node static function for adding data for lhs attribute and constant setting.

Fixes [Bug #15928]

History

#1 - 06/16/2019 02:28 PM - yugui (Yuki Sonoda)
s/lhs/rhs/

#2 - 06/16/2019 02:30 PM - chrisseaton (Chris Seaton)

There are specs that cover this, so at least it is how Ruby implementors understand that it is intended to be, and it's been this way for a decade or more.


#3 - 06/16/2019 02:37 PM - mame (Yusuke Endoh)

IMO, for expr::C = lhs, it should first evaluate expr and then do lhs because expr is left to lhs. But I'm unsure whether it should raise a TypeError when expr is not a Module. My personal expectation is that lhs is evaluated and then a TypeError is raised.

Anyway, the change will bring incompatibility. We need to consider whether it is worth fixing at the cost of incompatibility issues.
I submitted a pull request to fix this: https://github.com/ruby/ruby/pull/4450

Like @mame (Yusuke Endoh), I think this is a real bug. Constant assignment evaluation order should be consistent with attribute assignment evaluation order.

The spec referenced by @chrisseaton (Chris Seaton) was added by @headius (Charles Nutter) (https://github.com/ruby/spec/commit/34ea90b9b86d5290d546791c876e46b4c16b595) in response to a bug filed in JRuby’s JIRA bug tracker (no longer available). From JRuby’s 1.6.4 release notes (https://www.jruby.org/2011/08/22/jruby-1-6-4.html):

JRUBY-4925 Evaluation of constant happens before rhs of assignment (different to mri)

Like many specs, the added spec describes the actual behavior of CRuby at the time it was added, without regard for whether the behavior was intentional or not. So the fact that the spec exists is not a reason not to fix the bug. My pull request removes the spec.

Like many specs, the added spec describes the actual behavior of CRuby at the time it was added, without regard for whether the behavior was intentional or not.

If users depend on it and filed a bug, it becomes de facto expected behavior.

So the fact that the spec exists is not a reason not to fix the bug.

Indeed.

My pull request removes the spec.

No, the spec should be kept for older versions/other implementations, and we should have a spec for the new behavior.

If the behavior was a deliberate feature in older versions, I agree. If the behavior was not deliberate in old versions, and changing it is just fixing a bug (as it is in this case), I disagree. Simply because it took a long time for this bug to be fixed does not mean it should be treated differently from other bugs.

I believe all bug fixes that break specs should be treated this way (removing broken specs), and all features implemented that break specs be handled by adding appropriate version guards to existing specs. In either case, it’s fine but not required to add specs with appropriate version guards.

How do you know if it was deliberate?

FWIW I think it might be deliberate, I think the implementation is simpler and more consistent the way it is. For a.b.c, d.e.f = values we currently evaluate values, then a.b.c = values[0], then d.e.f = values[1] (correct me if I'm wrong). That's simple and there is no need to switch back-and-forth left and right (it's just RHS, then all of LHS in order).

We need to evaluate values before the actual assignments of course, and assignments can be method calls so I think it's just fine to execute all the assignments on the left part together and not half of it first, then values, then the foo= methods.

The current strategy also feels more efficient, because the suggested change actually needs to keep temporary values for the results of a.b and d.e additionally.

I think the implementation in TruffleRuby shows nicely how simple multiple assignment semantics are currently: https://github.com/oracle/truffleruby/blob/f2e97df99606ad382b4ba8f12729844ecf6c9b5f9/src/main/java/org/truffleruby/core/array/MultipleAssignmentNode.java#L61-L85

I think it would be significantly more complex with this change.

BTW that spec was added it seems before JIS 3017:2013, and JIS 3017:2013 has many differences with CRuby and many parts are not specified at all (e.g., class variables in singleton classes). So differences with JIS 3017:2013 are somewhat expected (not ideal though).
Simply because it took a long time for this bug to be fixed does not mean it should be treated differently from other bugs.

The key here is whether it would be backported. I'm confident that if this is merged it would not be backported (too big risk of incompatibility).

From https://github.com/ruby/ruby/pull/4450#discussion_r626213475 (sorry, I should have replied only here to keep it in one place).

If TruffleRuby/JRuby want to target Ruby 2.6/2.7/3.0 and want to make this change earlier, I don't think we should have a spec preventing it.

I'm extremely confident that JRuby, TruffleRuby and other Rubies would not change the behavior while they claim compatibility with Ruby <= 3.0. They want maximum compatibility with an existing release of Ruby, whether the semantics are ideal/match JIS 3017:2013 or not is typically much less relevant than compatibility with that CRuby version.

The fact there have been user bug report(s) there mean some code must rely on this.

Keeping the spec for previous versions codifies wrong behavior.

It's your interpretation that it is wrong behavior. The fact is Ruby <= 3.0 has that behavior and it's extremely unlikely to change (it's even impossible to change it for existing CRuby releases).

I believe all bug fixes that break specs should be treated this way (removing broken specs)

Maybe this way to explain is clearer: Removing the spec is equivalent to removing a test on a backport branch, it has the same effect, Ruby <= 3.0 is no longer tested for that case, and other Rubies targeting Ruby <= 3.0 too. Of course, removing a test on a backport branch is completely unacceptable (unless it's transient and causing CI issues maybe, doesn't apply here).

#8 - 05/05/2021 03:10 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-7:

jeremyevans0 (Jeremy Evans) wrote in #note-6:

If the behavior was a deliberate feature in older versions, I agree. If the behavior was not deliberate in old versions, and changing it is just fixing a bug (as it is in this case), I disagree.

How do you know if it was deliberate?

FWIW I think it might be deliberate, I think the implementation is simpler and more consistent the way it is. For a,b,c, d.e.f = values we currently evaluate values, then a.b.c = values[0], then d.e.f = values[1] (correct me if I'm wrong).

You're wrong since 50c54d40a81bb2a4794a6be5f1861152900b4fed (bug fix for #4443).

That's simple and there is no need to switch back-and-forth left and right (it's just RHS, then all of LHS in order).

Multiple assignment order should operate like single assignment order, and evaluate left-to-right. a.b = c evaluates a, c, then b=. Multiple assignment has been fixed to work the same way. The only reason multiple assignment evaluation order was broken for so long is that it was considered difficult to fix and nobody put in the effort to do so. The same issue applies to constant evaluation order (this issue).

We need to evaluate values before the actual assignments of course, and assignments can be method calls so I think it's just fine to execute all the assignments on the left part together and not half of it first, then values, then the foo= methods.

If we are doing to do that, we should change single assignment order. But it is fairly clear based on discussion in #4443 that the desired evaluation order is left-to-right.

The current strategy also feels more efficient, because the suggested change actually needs to keep temporary values for the results of a.b and d.e additionally.

Correct, there is a minor slowdown for evaluating left-to-right, measurable in microbenchmarks. It's unlikely to be significant in any real codebase. Obviously for constant assignment, that doesn't happen dynamically, so any slowdown is of no importance.

I think the implementation in TruffleRuby shows nicely how simple multiple assignment semantics are currently: https://github.com/oracle/truffleruby/blob/72e6c4e4a5a47a382c4ba811f729694ecacbf1b59/src/main/java/org/truffleruby/core/array/MultipleAssignmentNode.java#L61-L85

I think it would be significantly more complex with this change.
If you want to maintain compatibility with CRuby, you'll need to change it anyway, and when doing so, you might as well support constant assignment
the same way as attribute assignment.

Yes, it is much more complex. I'm guessing that's the reason that #4443 stayed open for so long.

BTW that spec was added it seems before JIS 3017:2013, and JIS 3017:2013 has many differences with CRuby and many parts are not
specified at all (e.g., class variables in singleton classes).
So differences with JIS 3017:2013 are somewhat expected (not ideal though).

Agreed. The primary reason to fix this is not for JIS compatibility, but for consistency with other assignment.

Simply because it took a long time for this bug to be fixed does not mean it should be treated differently from other bugs.

The key here is whether it would be backported.
I'm confident that if this is merged it would not be backported (too big risk of incompatibility).

I think the key issue is whether the spec documents behavior that is intended. If the spec documents behavior that was unintended, it should be
removed.

From https://github.com/ruby/ruby/pull/4450#discussion_r626213475 (sorry, I should have replied only here to keep it in one place).

I'm extremely confident that JRuby, TruffleRuby and other Rubies would not change the behavior while they claim compatibility with Ruby <= 3.0.
They want maximum compatibility with an existing release of Ruby, whether the semantics are ideal/match JIS 3017:2013 or not is typically
much less relevant than compatibility with that CRuby version.
The fact there have been user bug report(s) there mean some code must rely on this.

I tried to research this, but unfortunately the JRuby JIRA was removed. From my experience with CRuby, people report bugs when they notice
inconsistency that they think should be fixed, not because they are relying on the behavior.

Keeping the spec for previous versions codifies wrong behavior.

It's your interpretation that it is wrong behavior.

Not just mine. @yugui (Yuki Sonoda) and @mame (Yusuke Endoh) also feel it is wrong, and the issue is similar to #4443, in which @matz (Yukihiro
Matsumoto) confirms that left-to-right is the desired evaluation order. #4443 doesn't specifically discuss constant assignment, but it seems unlikely
that constant assignment was deliberately different than attribute assignment.

The fact is Ruby <= 3.0 has that behavior and it's extremely unlikely to change (it's even impossible to change it for existing CRuby releases).
I believe all bug fixes that break specs should be treated this way (removing broken specs)

Maybe this way to explain is clearer:
Removing the spec is equivalent to removing a test on a backport branch, it has the same effect, Ruby <= 3.0 is no longer tested for that case,
and other Rubies targeting Ruby <= 3.0 too.
Of course, removing a test on a backport branch is completely unacceptable (unless it's transient and causing CI issues maybe, doesn't apply
here).

In my opinion, we shouldn't test for behavior we consider wrong. The behavior was wrong in Ruby 1.8-3.0 (maybe even earlier). Yes, it took a long
time to fix. However, I don't think that changes the nature of things. This is a bug fix, not a feature change, and should be treated like other bug fixes.

#9 - 05/05/2021 03:24 PM - Eregon (Benoit Daloze)
- Related to Bug #4443: odd evaluation order in a multiple assignment added

#10 - 05/05/2021 03:44 PM - Eregon (Benoit Daloze)
Thank you for the context.
IMHO changing single assignment order is a lot simpler, and would bring consistency too.
I feel consistency also has rather low value here, multiple assignment is different to multiple single assignments, notably that all values are evaluated
together.
Evaluating values first is simple to explain, I think the new behavior isn't.

BTW, it sounds like MRuby always evaluates the value first: https://bugs.ruby-lang.org/issues/4443#note-8

08/28/2022
I could confirm it with:

```
$ mruby -e '{(p :a; a).b = p(:value)}'
:value
:a
$ mruby -e '{(p :a; a).b, v = p(:value)}'
:value
:a
```

MRuby’s behavior is simpler, consistent and more efficient.

@mame (Yusuke Endoh)
@ko1 (Koichi Sasada)
@yuqui (Yuki Sonoda) Are we sure we want to go ahead with the change in #4443 and this one? Or should we keep things as they are in 3.0, or as change single assignment like in MRuby?
This change does make Ruby slower and the implementation significantly more complicated, as Jeremy found in https://bugs.ruby-lang.org/issues/4443#note-20.
In other words, that change is costly in terms of implementation effort, risky in terms of compatibility (maybe not so much but feels weird to me to change in 3.1), and slower.

In my opinion, we shouldn’t test for behavior we consider wrong.

I believe we should if that behavior is exposed to users. And "wrong" is very much depending on personal opinions, so that sounds like a bad rule.

This is a bug fix, not a feature change, and should be treated like other bug fixes.

Like other bug fixes, none of them removes specs. They add a version guard.

A new example for the new behavior would be great, so we can clearly see what changed, but I won't fight for that in this issue (every other committer does it though AFAIK).

I recommend to see ruby/spec as a test suite shared between Ruby implementations (so it must support multiple Ruby versions), and to which all Ruby implementations can easily contribute (the name is confusing, I know, we all make mistakes).
Maybe it's easier to understand my point of view that way.
And since I'm the main maintainer/contributor of ruby/spec, I believe I get to define the vision and what is its purpose.

#11 - 05/05/2021 04:47 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-10:

```
This is a bug fix, not a feature change, and should be treated like other bug fixes.
```

Like other bug fixes, none of them removes specs. They add a version guard.

Stating absolutes without confirming them is risky. Looking through the commits to spec, c881678cd75432f47903a5d1d88b66a7a723eb023 just removed specs stating It is not a spec [...] of Ruby. I will grant that that most bug fixes add guards instead of removing specs.

A new example for the new behavior would be great, so we can clearly see what changed, but I won't fight for that in this issue (every other committer does it though AFAIK).

From a cursory analysis of the CRuby commit log, looking for commits that make modifications to either test or spec, the most common case is only files in test are modified. Modifications to spec are less frequent, with the vast majority of changes to spec coming from merges from ruby/spec, not direct commits. So every other committer does it though does not appear to be accurate.

I recommend to see ruby/spec as a test suite shared between Ruby implementations (so it must support multiple Ruby versions), and to which all Ruby implementations can easily contribute (the name is confusing, I know, we all make mistakes).
Maybe it's easier to understand my point of view that way.
And since I'm the main maintainer/contributor of ruby/spec, I believe I get to define the vision and what is its purpose.

If the purpose of ruby/spec is testing for observed behavior of CRuby, without consideration for whether the behavior is feature or bug, then it makes sense to keep the existing spec. I've updated the pull request to guard the current spec. I also added a basic spec for single constant assignment after this change.

#12 - 05/05/2021 05:12 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote in #note-11:

```
Stating absolutes without confirming them is risky. Looking through the commits to spec, c881678cd75432f47903a5d1d88b66a7a723eb023 just removed specs stating It is not a spec [...] of Ruby. I will grant that that most bug fixes add guards instead of removing specs.
```
Indeed. I think it holds for the vast majority though. That case is not fixing a bug in Ruby, it's handling a spec that does not work due to external changes.

So every other committer does it though does not appear to be accurate.

I meant that, AFAIK in most cases, when spec/ruby needs to be adapted when behavior changes the existing spec example(s) was kept under a version guard and a new spec example was added for the new behavior. I and other Ruby implementations devs really appreciate this, and that way the coverage is kept as good as before and there is a very clear test that demonstrates the changed behavior.

I've updated the pull request to guard the current spec. I also added a basic spec for single constant assignment after this change.

Thank you!

#13 - 05/07/2021 02:04 AM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote in #note-10:

@mame (Yusuke Endoh)@ko1 (Koichi Sasada)@yuqui (Yuki Sonoda) Are we sure we want to go ahead with the change in #4443 and this one? Or should we keep things as they are in 3.0, or as change single assignment like in MRuby?

In my opinion, it should be fixed eventually.

In principle, Ruby's evaluation is left-to-right. Almost all constructions including single assignment follow the principle. Multiple assignment and constant declaration were against it. I know that the evaluation semantics is challenging to implement, especially on the stack-based virtual machine. However, I don't think that it is good to violate the principle for a implementation-detail reason. I don't know the background of constant declaration, but in regard to multiple assignment (#4443), I faced the issue when I was writing a practical program to rotate a splay tree. Thus, the problem of evaluation order is not just conceptual but real to me.

That being said, I'm a little concerned about the incompatibility. If an actual issue is reported after preview release, we may have to proceed step by step, such as introducing migration path or something.

#14 - 05/20/2021 04:31 AM - nobu (Nobuyoshi Nakada)

- Description updated

#15 - 06/07/2021 03:41 PM - jeremyevans0 (Jeremy Evans)

This was discussed at the May 2021 developer meeting. @naruse (Yui NARUSE) is against breaking backwards compatibility for this in 3.1, so I'll repropose this after the release of 3.1.

#16 - 01/14/2022 02:59 AM - mame (Yusuke Endoh)

This was discussed at the dev-meeting, and @matz (Yukihiro Matsumoto) said it's time for a change. @jeremyevans0 (Jeremy Evans) Could you merge your pull request?

#17 - 01/14/2022 07:00 PM - jeremyevans (Jeremy Evans)

- Status changed from Open to Closed

Applied in changeset git|ca3d405242c722c8140944bda7278c2a9e5a7139.

Fix constant assignment evaluation order

Previously, the right hand side was always evaluated before the left hand side for constant assignments. For the following:

```ruby
lhs::C = rhs
```

rhs was evaluated before lhs, which is inconsistent with attribute
assignment (lhs.m = rhs), and apparently also does not conform to
JIS 3017:2013 11.4.2.2.3.

Fix this by changing evaluation order. Previously, the above
compiled to:

```
0000 putself                      (  l)[Li]
0001 opt_send_without_block <calldata!mid:rhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0003 dup
0004 putself
0005 opt_send_without_block <calldata!mid:lhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0007 setconstant :C
0009 leave
```

After this change:

```
0000 putself                      (  l)[Li]
0001 opt_send_without_block <calldata!mid:lhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0003 putself
0004 opt_send_without_block <calldata!mid:rhs, argc:0, FCALL|VCALL|ARGS_SIMPLE>
0006 swap
0007 topn                         1
0009 swap
0010 setconstant :C
0012 leave
```

Note that if expr is not a module/class, then a TypeError is not
raised until after the evaluation of rhs. This is because that
error is raised by setconstant. If we wanted to raise TypeError
before evaluation of rhs, we would have to add a VM instruction
for calling vm_check_if_namespace.

Changing assignment order for single assignments caused problems
in the multiple assignment code, revealing that the issue also
affected multiple assignment. Fix the multiple assignment code
so left-to-right evaluation also works for constant assignments.

Do some refactoring of the multiple assignment code to reduce
duplication after adding support for constants. Rename struct
masgn_atrasgn to masgn_lhs_node, since it now handles both
constants and attributes. Add add_masgn_lhs_node static function
for adding data for lhs attribute and constant setting.

Fixes [Bug #15928]