Ruby master - Feature #11660

a falsy value (similar to js undefined) that facilitates forwarding of default arguments

11/05/2015 08:36 PM - bughit (bug hit)

Status: Open
Priority: Normal
Assignee: 
Target version: 

Description
I'll call it "missing" here. Consider the following scenario:

```ruby
def foo(default1 = some_expression)
  end

def bar(default1 = some_expression)
  foo default1
  end

def foobar(default1 = some_expression)
  bar default1
  end

if you had "missing":

  def foo(default1 = some_expression)
    end

  def bar(default1 = missing)
    foo default1
    end

  def foobar(default1 = missing)
    bar default1
    end

missing passed as arg would be ignored (as if it wasn't passed at all)
and you wouldn't have to repeat the default value expression in every method

I believe that's how undefined works in js6 with respect to default args
```

History

#1 - 11/01/2019 07:02 PM - bughit (bug hit)

Compared to javascript default arg values are far less useful in ruby because of the lack of this forwarding capability.

This issue is 4 years old, can ruby core at least consider this?

#2 - 11/01/2019 07:21 PM - jeremyevans0 (Jeremy Evans)

Other than arity, it seems like you would want to use the new argument forwarding syntax:

```ruby
def foo(default1 = some_expression)
  end

def bar(...) 
  foo(...) 
  end

def foobar(...) 
  bar(...) 
  end

The major issue with adding this support is what the value of this is:

  def foobar(default1 = missing)
```

05/10/2020
Other than arity, it seems like you would want to use the new argument forwarding syntax

I didn't know about new syntax, is that in 2.7? However, I don't think that's general enough. In javascript you can forward individual arguments to an arbitrary depth and the default value is computed once at the end.

```javascript
function f1(a = defaultValueExpression) {
}

function f2({b = 1, c} = {}) {
    f1(c);
}

function f3(d, e) {
    f2({b: d, c: e});
}

f3(1)
// or
f3(1, undefined)
```

the missing/undefined second arg to f3 invocation flows all the way down to f1 where it triggers default value computation

You can't do this in ruby, to accomplish the equivalent you'd have to stop using intended default value expressions, forward nils, and then compute the default value in the body of the method.

```ruby
def f1(a = nil)
    a = default_value_expression if a.nil?
end
def f2(b: 1, c: nil)
    f1(c)
end
def f3(d, e = nil)
    f2(b: d, c: e)
end

f3(1)
```

So in ruby the default value feature is close to being demoware. In practice, when composing methods, you end up reverting to the above.

The major issue with adding this support is what the value of this is

The value would be missing. Adding something like missing to the language would need to be coupled with a null coalescing operator which would cover both nil and missing (but not false). Something like `??` and `??=`

#4 - 11/01/2019 09:55 PM - jeremyevans0 (Jeremy Evans)
bughit (bug hit) wrote:

The value would be missing. Adding something like missing to the language would need to be coupled with a null coalescing operator which would cover both nil and missing (but not false). Something like `??` and `??=`

Adding a new keyword and operator seems like a high cost for at best a very minor benefit, in my opinion. I'm definitely against adding this.

#5 - 11/01/2019 11:55 PM - bughit (bug hit)
It's conceivable that it can be done without a keyword, there are other way to refer to special values.

However what do you have against a null coalescing operator? It's needed regardless of this issue, `||` and `||=` are inadequate because of false.

#6 - 11/02/2019 12:32 AM - alanwu (Alan Wu)
Since the beginning of time (correct me if I'm wrong), the two falsy values in Ruby has been false and nil, period. Adding another falsy value is a big change. The missing/undefined second arg to f3 invocation flows all the way down to f1 where it triggers default value computation. You can't do this in ruby.

You can do it if all the methods in the chain use splat. In 2.7 ... also works:

```ruby
def f3( foo = 500, bar = nil)
  p [ foo, bar]
end

def f2(*args)
  f3(*args)
end

def f1(*args)
  f2(*args)
end
f1(:bar) # [500, :bar]
```

https://wandbox.org/permlink/3xAq9B2wQk8odafG

Default args have been a feature for a long time, the fact there hasn't been a proposal similar to yours suggests to me that it's not a feature in high demand.

#7 - 11/02/2019 10:06 AM - decuplet (Nikita Shilnikov)
For this purpose I built a special value a while ago. It has been working fine all this time. I don't think there's a need in adding such special-cased value to the language. It would only slightly improve the experience over a hand-crafted ten-lines-of-code solution. The whole story of passing arguments in Ruby is already complicated enough.

#8 - 11/03/2019 06:57 PM - bughit (bug hit)

For this purpose I built a special value a while ago.

What purpose? Surely you understand that your special value, having no compiler support, does not trigger default value computations.

You are making my point. Your Undefined is not a real, intended default value, but a signal to compute the real default value in the body of the method after an explicit check for this signal.

This under-baked design of the default value feature is the point of this bug.

#9 - 11/04/2019 10:24 AM - decuplet (Nikita Shilnikov)
The "penalty" I pay for this is one line of code like period = Undefined.default(period, :day). Given this doesn't occur very often I don't see how "compiler support" would be beneficial.

#10 - 11/04/2019 03:22 PM - Dan0042 (Daniel DeLorme)
alanwu (Alan Wu) wrote:

Since the beginning of time (correct me if I'm wrong), the two falsy values in Ruby has been false and nil, period. Adding another falsy value is a big change.

I Agree. Not saying there's no benefit, but this is too big a change for too small a benefit. Usually nil will play the role of the undefined value, so it might be nice if a method definition allowed something like def foo(v ||= 42). But nowadays we have keyword arguments which do exactly what is needed in this case.

```ruby
def f2(x: 2)
  x
end

def f1(y:1, **kw)
  y + f2(**kw)
end
f1(y: 3) #=> 5
```
The "penalty" I pay for this is one line of code

The penalty the language pays is it has a demoware, checklist item, default value feature, where in any non-trivial scenario you have to produce the default value twice, first the intermediate default value through the default value expression, then the real default value in the body. You are not doing ruby any favors by defending this "design".

bughit (bug hit) wrote:

The penalty the language pays is it has a demoware, checklist item, default value feature, where in any non-trivial scenario you have to produce the default value twice, first the intermediate default value through the default value expression, then the real default value in the body.

There are many non-trivial scenarios where you don't need to do that (or we can have a no-true-Scotsman argument about the definition of non-trivial scenario). Most method calls do not involve calling methods that delegate arguments to another method where both methods have the same optional argument with the same default value. In the cases where that happens, the general practice is just to duplicate the default value in both methods, which is not usually an issue as default values are usually simple values or method calls.

The only time you really want to have the "real default value" in the body is in fairly complex cases, and in those cases, the default argument values are usually set to nil. This is easy to understand and has worked well for Ruby for a long time. Ruby having a single nil, instead of JavaScript's null and undefined, is a net benefit, in my opinion.

You are not doing ruby any favors by defending this "design".

You are not doing Ruby any favors by making a mountain out of a molehill. The popularity of an idea is not always a proxy to its merit, but if nobody else thinks the benefits of a proposal exceed the costs, maybe it isn't worth implementing.

Ruby having a single nil, instead of JavaScript's null and undefined, is a net benefit, in my opinion.

undefined signals the absence of a value and makes possible composable default value design, where missing arguments can be forwarded and still trigger default value computation, which is a clear win for javascript.

The popularity of an idea is not always a proxy to its merit, but if nobody else thinks the benefits of a proposal exceed the costs, maybe it isn't worth implementing.

Ok got it. Popularity does not always equal merit, but often it does and this must be one of those times, because of course it has no merit and it's unpopular. Ergo ipso facto, it has no merit.