Refinements only get "used" once in loop

11/17/2015 11:08 PM - danielpclark (Daniel P. Clark)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
<tr>
<td>Backport:</td>
<td>2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN</td>
</tr>
</tbody>
</table>

Description

Same results on Ruby 2.2.2 through Ruby 2.3.0dev (2015-11-18 trunk 52625) [x86_64-linux]

I wrote a benchmark for testing different ways of implementing a uniq method and I chose to do it using refinements. I looped over the results in the benchmark and refined the method with 6 different refinements which each worked. But the next iteration when using is called it doesn't re-refine with previously used refinements.

Example benchmark output on first iteration:

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Precision</th>
<th>Mean</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array.new(200) !self.dup.uniq!</td>
<td>1.770000</td>
<td>0.010000</td>
<td>1.780000</td>
<td>( 1.778248)</td>
</tr>
<tr>
<td>Array.new(200) == uniq.length</td>
<td>1.860000</td>
<td>0.000000</td>
<td>1.860000</td>
<td>( 1.866862)</td>
</tr>
<tr>
<td>Array.new(200) == uniq.sort</td>
<td>2.580000</td>
<td>0.010000</td>
<td>2.590000</td>
<td>( 2.584515)</td>
</tr>
<tr>
<td>Array.new(200) each index</td>
<td>41.450000</td>
<td>0.080000</td>
<td>41.530000</td>
<td>( 41.626149)</td>
</tr>
<tr>
<td>Array.new(200) combination(2)</td>
<td>23.460000</td>
<td>0.060000</td>
<td>23.520000</td>
<td>( 23.568865)</td>
</tr>
<tr>
<td>Array.new(200) == self.uniq</td>
<td>1.900000</td>
<td>0.010000</td>
<td>1.910000</td>
<td>( 1.909466)</td>
</tr>
</tbody>
</table>

After that the same methods did not get refined.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Precision</th>
<th>Mean</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array.new(210) !self.dup.uniq!</td>
<td>1.990000</td>
<td>0.000000</td>
<td>1.990000</td>
<td>( 2.004269)</td>
</tr>
<tr>
<td>Array.new(210) == uniq.length</td>
<td>2.030000</td>
<td>0.010000</td>
<td>2.040000</td>
<td>( 2.032602)</td>
</tr>
<tr>
<td>Array.new(210) == uniq.sort</td>
<td>1.990000</td>
<td>0.000000</td>
<td>1.990000</td>
<td>( 1.999509)</td>
</tr>
<tr>
<td>Array.new(210) each index</td>
<td>1.990000</td>
<td>0.010000</td>
<td>2.000000</td>
<td>( 2.000181)</td>
</tr>
<tr>
<td>Array.new(210) combination(2)</td>
<td>2.000000</td>
<td>0.000000</td>
<td>2.000000</td>
<td>( 2.001059)</td>
</tr>
<tr>
<td>Array.new(210) == self.uniq</td>
<td>2.000000</td>
<td>0.010000</td>
<td>2.010000</td>
<td>( 2.009117)</td>
</tr>
</tbody>
</table>

I found no way to inspect what code was being used as refinements currently don't allow introspection (I have read the Refinement Specs [https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec](https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec)). But I figure if I wanted to see what the refinement was I could write an additional method to document the current refinement in use.

```ruby
module BooleanUniqWithEqualLength
  refine Array do
    def uniq?
      self.length == self.uniq.length
    end

    def which_refinement_uniq?
      "self.length == self.uniq.length"
    end

end
```
But introspection is not the issue I'm raising here. The issue is that you can only use the refinement once within the scope of the loop. Look at the benchmark results and see the first 6 are correct, and the following 3 times the output is lying about what refinement is being used.

Here's the full benchmark code:

```ruby
require 'securerandom'
require 'benchmark'

# written to allow 65,536 unique array items
array_of_x = lambda { |x| SecureRandom.hex(x*2).scan(/.../) }

module Refinements
  module BooleanUniqWithDupUniqBang
    refine Array do
      def uniq!
        !self.dup.uniq!
      end
    end
  end

  module BooleanUniqWithEqualLength
    refine Array do
      def uniq?
        self.length == self.uniq.length
      end
    end
  end

  module BooleanUniqWithEqualSort
    refine Array do
      def uniq?
        self.sort == self.uniq.sort
      end
    end
  end

  module BooleanUniqWithEachIndex
    refine Array do
      def uniq?
        self.each_with_index { |a,b|
          self.each_with_index { |c,d|
            next if b == d
            return false if a == c
          }
        }
        true
      end
    end
  end

  module BooleanUniqWithCombination
    refine Array do
      def uniq?
        self.combination(2).each { |a,b| return false if a == b }
        true
      end
    end
  end

  module BooleanUniqWithUniq
    refine Array do
      def uniq?
        self == self.uniq
      end
    end
  end
end
```

05/10/2020
bench_reps = 10_000

bench = Benchmark.benchmark("\nTesting various ways of implementing :uniq? on Array\n(smaller numbers are better)\n\n", 34) do |x|
  # Note doing anymore than one test per test type seems to wash the results into all things being e
  # Only the first test gives realistic numbers.
  [500].each do |qty|
    x.report("Array.new(#{qty}) !self.dup.uniq!") do
      using Refinements::BooleanUniqWithDupUniqBang
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end

    x.report("Array.new(#{qty}) == uniq.length") do
      using Refinements::BooleanUniqWithEqualLength
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end

    x.report("Array.new(#{qty}) == uniq.sort") do
      using Refinements::BooleanUniqWithEqualSort
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end

    x.report("Array.new(#{qty}) each index") do
      using Refinements::BooleanUniqWithEachIndex
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end

    x.report("Array.new(#{qty}) combination(2)") do
      using Refinements::BooleanUniqWithCombination
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end

    x.report("Array.new(#{qty}) == self.uniq") do
      using Refinements::BooleanUniqWithUniq
      bench_reps.times do
        array_of_x.(qty).uniq?
      end
    end
  end
end

History
#1 - 11/17/2015 11:15 PM - danielpclark (Daniel P. Clark)
Change the
[500].each do |qty|
For the same benchmark test.

#2 - 11/26/2015 08:20 AM - shugo (Shugo Maeda)
- Status changed from Open to Assigned
- Assignee set to matz (Yukihiro Matsumoto)

Daniel P. Clark wrote:

I wrote a benchmark for testing different ways of implementing a uniq method and I chose to do it using refinements. I looped over the results in the benchmark and refined the method with 6 different refinements which each worked. But the next iteration when using is called it doesn't re-refine with previously used refinements.

Once refinements are activated by using in a specific order, the precedence of the refinements cannot be changed, because Refinements are not designed for such dynamic use.

What do you think, Matz?

#3 - 04/13/2016 08:19 PM - danielpclark (Daniel P. Clark)
According to John (who commented on my blog on this issue) this is a Dynamic Dispatch issue. And the following example may be a related.

```ruby
module Moo
  refine Fixnum do
    def to_s
      "moo"
    end
  end
end
class A
  using Moo
  def a
    [1, 2, 3].map { |x| x.to_s }
  end
  def b
    [1, 2, 3].map(&:to_s)
  end
end
A.new.a
# => ["moo", "moo", "moo"]
A.new.b
# => ["1", "2", "3"]
```

#4 - 04/14/2016 02:03 AM - sawa (Tsuyoshi Sawada)
Daniel P. Clark wrote:

According to John (who commented on my blog on this issue) this is a Dynamic Dispatch issue. And the following example may be a related.

I had made a feature request #12079 (later than this post) to allow refinements to be effective in such cases. It can be a way to think about this issue.

#5 - 04/14/2016 02:45 AM - danielpclark (Daniel P. Clark)
Tsuyoshi Sawada wrote:

I had made a feature request #12079 (later than this post) to allow refinements to be effective in such cases.

I like it. One note: Your code examples don't have a lexically scoped block. For them to work in Ruby 2.2.3 in the global scope as you're showing you'd need a begin/end block. Works in Ruby 2.2.3 but not 2.3.0. I believe as of Ruby 2.3 it may only be done within a class definition.

```ruby
module Moo
  refine Fixnum do
    def to_s
      "moo"
    end
  end
end
```
begin # valid Ruby 2.2.3 and NOT Ruby 2.3
    using Moo
    1.to_s
end
# => "moo"