Ruby trunk - Feature #4085

Refinements and nested methods

11/24/2010 10:12 PM - shugo (Shugo Maeda)

Status:	Closed	
Priority:	Normal	
Assignee:	shugo (Shugo Maeda)	
Target version:	2.0.0	

Description

=begin
As I said at RubyConf 2010, I'd like to propose a new features called
"Refinements."

Refinements are similar to Classboxes. However, Refinements doesn't
support local rebinding as mentioned later. In this sense,
Refinements might be more similar to selector namespaces, but I'm not
sure because I have never seen any implementation of selector
namespaces.

In Refinements, a Ruby module is used as a namespace (or classbox) for
class extensions. Such class extensions are called refinements. For
example, the following module refines Fixnum.

module MathN
refine Fixnum do
def /(other) quo(other) end
end
end

Module#refine(klass) takes one argument, which is a class to be
extended. Module#refine also takes a block, where additional or
overriding methods of klass can be defined. In this example, MathN
refines Fixnum so that 1 / 2 returns a rational number (1/2) instead
of an integer 0.

This refinement can be enabled by the method using.

class Foo
using MathN

```
def foo
  p 1 / 2
end
```

end

f = Foo.new
f.foo #=> (1/2)
p 1 / 2

In this example, the refinement in MathN is enabled in the definition
of Foo. The effective scope of the refinement is the innermost class,
module, or method where using is called; however the refinement is not
enabled before the call of using. If there is no such class, module,
or method, then the effective scope is the file where using is called.
Note that refinements are pseudo-lexically scoped. For example,
foo.baz prints not "FooExt#bar" but "Foo#bar" in the following code:

class Foo
def bar
puts "Foo#bar"
end

```
 def baz
   bar
 end

end

module FooExt
refine Foo do
def bar
puts "FooExt#bar"
end
end
end

module Quux
using FooExt

 foo = Foo.new
 foo.bar  # => FooExt#bar
 foo.baz  # => Foo#bar

end
```

Refinements are also enabled in reopened definitions of classes using
refinements and definitions of their subclasses, so they are
pseudo-lexically scoped.

```
class Foo
using MathN
end

class Foo
# MathN is enabled in a reopened definition.
p 1 / 2  #=> (1/2)
end

class Bar < Foo
# MathN is enabled in a subclass definition.
p 1 / 2  #=> (1/2)
end
```

If a module or class is using refinements, they are enabled in
module_eval, class_eval, and instance_eval if the receiver is the
class or module, or an instance of the class.

```
module A
using MathN
end
class B
using MathN
end
MathN.module_eval do
p 1 / 2  #=> (1/2)
end
A.module_eval do
p 1 / 2  #=> (1/2)
end
B.class_eval do
p 1 / 2  #=> (1/2)
end
B.new.instance_eval do
p 1 / 2  #=> (1/2)
end
```

Besides refinements, I'd like to propose new behavior of nested methods.
Currently, the scope of a nested method is not closed in the outer method.

```
def foo
def bar
puts "bar"
end
bar
end
foo  #=> bar
bar  #=> bar
```

In Ruby, there are no functions, but only methods. So there are no right places where nested methods are defined. However, if refinements are introduced, a refinement enabled only in the outer method would be the right place. For example, the above code is almost equivalent to the following code:

```
def foo
klass = self.class
m = Module.new {
refine klass do
def bar
puts "bar"
end
end
}
using m
bar
end
foo  #=> bar
bar  #=> NoMethodError
```

The attached patch is based on SVN trunk r29837.
=end

Related issues:

Related to Ruby trunk - Bug #7271: Refinement doesn't seem lexical	**Closed**	**11/04/2012**
Related to Ruby trunk - Feature #7251: using usings in usinged Module	**Closed**	**10/31/2012**
Has duplicate Ruby trunk - Feature #6287: nested method should only be visibl...	**Closed**	**04/13/2012**

History

#1 - 11/24/2010 10:28 PM - ehuard (Elise Huard)

=begin
+1 definitely
=end

#2 - 11/24/2010 10:38 PM - rkh (Konstantin Haase)

=begin
+1, can't wait to see this merged into trunk.
=end

#3 - 11/24/2010 10:45 PM - dohzya (Etienne Vallette d'Osia)

=begin
+1 for refinements.

What about the problems you showed during your talk ?

For nested functions, I suppose it is better to transform them into lambdas,
but the concept is nice.
=end

#4 - 11/24/2010 11:00 PM - mr-rock (Javier Cicchelli)

=begin
+1 This is a neat solution. I like it ;)
=end

#5 - 11/24/2010 11:47 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33322] [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Wed, 24 Nov 2010 22:12:24 +0900, Shugo Maeda redmine@ruby-lang.org writes:

|Feature #4085: Refinements and nested methods
|http://redmine.ruby-lang.org/issues/show/4085

My +1.

However I'd like to see Ko1's consent before checking in. Last time I
talked with him about this issue, he wanted to consider this for a
week. That was last Friday. So we need to wait for a few more days.

```
                              matz.
```

=end

#6 - 11/25/2010 12:23 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/24 Etienne Vallette d'Osia redmine@ruby-lang.org:

> +1 for refinements.

Thank you.

> What about the problems you shown during your talk ?

No progress yet. So details may be changed after it is merged into trunk.

> For nested function, I suppose it is better to transform them into lambdas,

I think it's a (maybe too) big change because it means that real
functions are introduced into Ruby.
I like the fact that there are no functions (except lambda) in Ruby.
Even def at the top-level defines not a function but a method. It's
consistent as an object-oriented language.

Anyway, if my proposal about nested methods is not acceptable, I
withdraw it and propose only refinements.

--
Shugo Maeda

=end

#7 - 11/25/2010 12:34 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/24 Yukihiro Matsumoto matz@ruby-lang.org:

> |Feature #4085: Refinements and nested methods
> |http://redmine.ruby-lang.org/issues/show/4085
>
> My +1.

Thank you.

> However I'd like to see Ko1's consent before checking in. Last time I
> talked with him about this issue, he wanted to consider this for a
> week. That was last Friday. So we need to wait for a few more days.

I met ko1 last Thursday, and he agreed with my proposal at that time.
I hope he has not changed his mind.

I know that he doesn't like my implementation, where I have added a
new rb_control_frame_t member, which is equivalent to
ruby_frame->last_class in Ruby 1.8.
If he can implement refinements without the new member, it's OK for me:)

--
Shugo Maeda

=end

#8 - 11/25/2010 01:41 AM - pragdave (Dave Thomas)

=begin
module Quux
using FooExt

```
foo = Foo.new
foo.bar   # => FooExt#bar
foo.baz   # => Foo#bar
```

end

This behavior makes me nervous—I can see arguments for it, but at the same time I can see in leading to problems, particularly in well structured classes where a large set of behaviors is defined in terms of one method. I'm sure the syntax below is wrong, but look at the spirit of it.

module DoubleEach
refine Array do
def each
super do |val|
yield 2*val
end
end
end
end

using DoubleEach

[1, 2, 3].each {|v| p v }   #=> 2, 4, 6

[1, 2, 3].min   #=> 1

That would be surprising to me, as I'd expect the behavior of all the Enumerable methods, which depend on each, to change if I change the behavior of each.
=end

#9 - 11/25/2010 01:01 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/11/24 Shugo Maeda redmine@ruby-lang.org:

> As I said at RubyConf 2010, I'd like to propose a new features called
> "Refinements."

Basically I agree with your proposal, but I think that some
discussion is needed.

- Your patch is too big. Could you separate it to some parts?
  It is hard (for me) to review it.

- Your patch adds `klass' to stack frame, which may cause
  big performance degradation. We should check benchmark
  result (though I'm not so concerned).

- API design. Why did you select:

  module MathN
  refine(Fixnum) do
  def /(other) quo(other) end
  end
  end
  using MathN

and not select others, such as:

```
module MathN
refine(Fixnum)
def /(other) quo(other) end
end
using MathN
```

or:

```
MathN = refine(Fixnum) do
def /(other) quo(other) end
end
using MathN
```

IMO, it will be more natural to provide this feature as new
constract with new syntax, instead of Module's methods.

- Is it intended to reject refining module methods?

```
module ComplexExt
refine(Math) do
def sqrt(x)
(x >= 0 ? 1 : Complex::I) * super(x.abs)
end
end
end
#=> wrong argument type Module (expected Class) (TypeError)
```

--
Yusuke Endoh mame@tsg.ne.jp

=end

#10 - 11/25/2010 01:14 PM - shyouhei (Shyouhei Urabe)

=begin
(2010/11/25 13:01), Yusuke ENDOH wrote:

- Your patch is too big. Could you separate it to some parts? It is hard (for me) to review it.

Shugo, push it to github.

- Your patch adds `klass' to stack frame, which may cause big performance degradation. We should check benchmark result (though I'm not so concerned).

According to Shugo's presentation performance impact is up to 3% or so, which
is why Ko1 is not against it. It is almost no slowdown.

=end

#11 - 11/25/2010 01:35 PM - shugo (Shugo Maeda)

=begin
Hi,

Thanks for you comment.

2010/11/25 Dave Thomas redmine@ruby-lang.org:

```
module Quux
  using FooExt

  foo = Foo.new
  foo.bar  # => FooExt#bar
  foo.baz  # => Foo#bar
end
```

This behavior makes me nervous—I can see arguments for it, but at the same time I can see in leading to problems, particularly in well structured classes where a large set of behaviors is defined in terms of one method. I'm sure the syntax below is wrong, but look at the spirit of it.

```
module DoubleEach
```

```
    refine Array do
      def each
        super do |val|
          yield 2*val
        end
      end
    end
  end

  using DoubleEach

  [ 1, 2, 3 ].each {|v| p v }   #=> 2, 4, 6

  [ 1, 2, 3 ].min   #=> 1
```

The syntax right and currently [1,2,3].min returns 2.
However, It's a bug (methods implemented by C behave wrong),
and I think it should return 1 as you showed above.

> That would be surprising to me, as I'd expect the behavior of all the Enumerable methods, which depend on each, to change if I change the behavior of each.

However, some other person might expect the original behavior of each.
For example,

class Foo
using DoubleEach

```
def bar
  SomeOtherModule.do_something # DoubleEach may break this method
end
```

end

If SomeOtherModule.do_something is implemented by someone else, and
Array#each is used in do_something, DoubleEach breaks the code.
If you'd like to change the behavior of callees, you can use monky
patching or singleton methods instead of refinements.

--
Shugo Maeda

=end

#12 - 11/25/2010 01:55 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/25 Yusuke ENDOH mame@tsg.ne.jp:

> > As I said at RubyConf 2010, I'd like to propose a new features called
> > "Refinements."
>
> Basically I agree with your proposal, but I think that some
> discussion is needed.
>
> - Your patch is too big. Could you separate it to some parts?
>   It is hard (for me) to review it.

Shyouhei suggested github, but my github account has no enough disk
space, so I put the git format-patch result at the following site:

http://shugo.net/tmp/refinement-r29837-20101124.zip (ZIP archive)
http://shugo.net/tmp/refinement-r29837-20101124/ (Expanded directory)

However, I'm afraid that there are too many patches, some of which
have been already reverted.

> - Your patch adds `klass' to stack frame, which may cause
>   big performance degradation. We should check benchmark

result (though I'm not so concerned).

I have run "make benchmark" 5 times, and it shows that the modified version is slower than average 2.5% than the original version.

The benchmark result is available at:

http://shugo.net/tmp/refinement-benchmark-20101107.csv

Additional tests are welcome.

- API design. Why did you select:

```
module MathN
  refine(Fixnum) do
    def /(other) quo(other) end
  end
end
using MathN
```

and not select others, such as:

```
module MathN
  refine(Fixnum)
  def /(other) quo(other) end
end
using MathN
```

or:

```
MathN = refine(Fixnum) do
  def /(other) quo(other) end
end
using MathN
```

Because I'd like to allow multiple refinements in one module, for example, in the following case:

```
module MyXmlFormat
refine Integer do
def to_xml; ...; end
end
```

```
refine String do
  def to_xml; ...; end
end

refine Hash do
  def to_xml; ...; end
end
...
```

```
end
```

In this case, I prefer the above syntax to your proposals.

IMO, it will be more natural to provide this feature as new constract with new syntax, instead of Module's methods.

First, I consider it, but I wouldn't like to introduce new keywords.

- Is it intended to reject refining module methods?

```
module ComplexExt
  refine(Math) do
    def sqrt(x)
      (x >= 0 ? 1 : Complex::I) * super(x.abs)
    end
  end
end
#=> wrong argument type Module (expected Class) (TypeError)
```

It's limitation of the current implementation.
However, I don't think it's critical, because If you'd like to refine
a module, you can refine a class which includes the module instead.

--
Shugo Maeda

=end

#13 - 11/25/2010 02:25 PM - shyouhei (Shyouhei Urabe)

=begin
(2010/11/25 13:55), Shugo Maeda wrote:

> > - Your patch is too big. Could you separate it to some parts? It is hard (for me) to review it.
>
> Shyouhei suggested github, but my github account has no enough disk
> space, so I put the git format-patch result at the following site:

I've just uploaded in place of you. Here it is:

https://github.com/shyouhei/ruby/compare/shugo%2Frefinement

=end

#14 - 11/25/2010 02:29 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/25 Urabe Shyouhei shyouhei@ruby-lang.org:

> > Shyouhei suggested github, but my github account has no enough disk
> > space, so I put the git format-patch result at the following site:
>
> I've just uploaded in place of you. Here it is:
>
> https://github.com/shyouhei/ruby/compare/shugo%2Frefinement

Thank you.

--
Shugo Maeda

=end

#15 - 11/25/2010 06:46 PM - rkh (Konstantin Haase)

=begin
On Nov 25, 2010, at 05:55 , Shugo Maeda wrote:

> > IMO, it will be more natural to provide this feature as new
> > constract with new syntax, instead of Module's methods.
>
> First, I consider it, but I wouldn't like to introduce new keywords.

Plus that way it would be possible to stub the refine method and write code that does work on earlier ruby versions. Imagine RSpec switching to refinements in order to avoid global namespace pollution. It would be rather easy to implement a Module#refine that does in fact pollute the global namespace as a fall back. If new syntax is added, this wouldn't be possible.

Konstantin

=end

#16 - 11/25/2010 08:07 PM - mame (Yusuke Endoh)

=begin
Hi,

Thank you for your reply!

2010/11/25 Shugo Maeda shugo@ruby-lang.org:

> However, I'm afraid that there are too many patches, some of which
> have been already reverted.

Please fair them :-)

> - Your patch adds `klass' to stack frame, which may cause big performance degradation. We should check benchmark result (though I'm not so concerned).

> I have run "make benchmark" 5 times, and it shows that the modified
> version is slower than average 2.5% than the original version.

Sounds good!

> - API design. Because I'd like to allow multiple refinements in one module, for example, in the following case:

```
module MyXmlFormat
refine Integer do
def to_xml; ...; end
end

refine String do
def to_xml; ...; end
end

refine Hash do
def to_xml; ...; end
end
...
end
```

After my short trial of this feature, I'd like an API to do
refine' andusing' at once without explicit module, such as:

```
using_refine(Fixnum) do
def /(other) quo(other) end
end
```

(equivalent to)

```
using(Module.new do
refine(Fixnum) do
def /(other) quo(other) end
end
end)
```

Note that we cannot define `using_define' by ourselves because
of lexical scope limitation:

```
def using_refine(klass, &blk)
using(Module.new { refine(klass, &blk) })
end

using_refine(Fixnum) do
def /(other) quo(other) end
end

p 1 / 2  #=> 0
```

I guess that this `using_refine' is useful itself (though its
name is arguable). In addition, it allows us to write
MyXmlFormat as follows:

```
module MyXmlFormat
using_refine(Integer) do
def to_xml; ...; end
end
```

```
using_refine(String) do
  def to_xml; ...; end
end

using_refine(Hash) do
  def to_xml; ...; end
end
```

end

> > IMO, it will be more natural to provide this feature as new
> > constract with new syntax, instead of Module's methods.
>
> First, I consider it, but I wouldn't like to introduce new keywords.

I thought so, but I think that this feature deserves new
keywords because it is big evolution of Ruby's OO paradigm
(involving semantics change).
However, we should discuss this topic (new keyword) towards
2.0. Module's methods are not bad, as a part of reflection
features (such as Module#define_method for `def' keyword).

> > - Is it intended to reject refining module methods? It's limitation of the current implementation.

If so, it may be good to raise a NotImplementedError.

> However, I don't think it's critical, because If you'd like to refine
> a module, you can refine a class which includes the module instead.

Do you mean:

include Math

$toplevel = self
module ComplexExt
def sqrt(x)
(x >= 0 ? 1 : Complex::I) * super(x.abs)
end
refine(class << Math; self; end) { include ComplexExt }
refine(class << $toplevel; self; end) { include ComplexExt }
end

using ComplexExt

p sqrt(-4)

?

--
Yusuke Endoh mame@tsg.ne.jp

=end

#17 - 11/25/2010 10:49 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/25 Yusuke ENDOH mame@tsg.ne.jp:

> > However, I'm afraid that there are too many patches, some of which
> > have been already reverted.
>
> Please fair them :-)

I have tried "git rebase -i", but these commits are hard to reorder
because they are related to each other. So I have to separate the
whole patch manually.

How about to separate them into the following three patches?

1. changes of control frames and method lookup
2. Refinements support
3. nested methods support

> After my short trial of this feature, I'd like an API to do
> refine' andusing' at once without explicit module, such as:
>
> ```
>  using_refine(Fixnum) do
>    def /(other) quo(other) end
>  end
> ```
>
> (equivalent to)
>
> ```
>  using(Module.new do
>    refine(Fixnum) do
>      def /(other) quo(other) end
>    end
>  end)
> ```

I forgot to mention that refinements are enabled in class or module
definitions where Module#refine is called.

class Foo
refine Fixnum do
def /(other) quo(other) end
end

```
p 1 / 2  #=> (1/2)
```

end

Do we need Kernel#refine?

> I guess that this `using_refine' is useful itself (though its
> name is arguable). In addition, it allows us to write
> MyXmlFormat as follows:
>
> ```
>  module MyXmlFormat
>    using_refine(Integer) do
>      def to_xml; ...; end
>    end
>
>    using_refine(String) do
>      def to_xml; ...; end
>    end
>
>    using_refine(Hash) do
>      def to_xml; ...; end
>    end
>  end
> ```

I guess Module#refine works the same as using_refine in this case.

> > > IMO, it will be more natural to provide this feature as new
> > > constract with new syntax, instead of Module's methods.
>
> > First, I consider it, but I wouldn't like to introduce new keywords.
>
> I thought so, but I think that this feature deserves new
> keywords because it is big evolution of Ruby's OO paradigm
> (involving semantics change).
> However, we should discuss this topic (new keyword) towards
> 2.0. Module's methods are not bad, as a part of reflection
> features (such as Module#define_method for `def' keyword).

I don't think we need new keywords even if it is a big change,
because some essential features such as module inclusion
have no keyword in Ruby.

> > - Is it intended to reject refining module methods?
> It's limitation of the current implementation.

> If so, it may be good to raise a NotImplementedError.

Agreed.

> > However, I don't think it's critical, because If you'd like to refine
> > a module, you can refine a class which includes the module instead.

> Do you mean:

I guess it's unlikely that you need refine module functions in
real-world applications.
Why don't you simply override module functions?

```
include Math
def sqrt(x)
  (x >= 0 ? 1 : Complex::I) * super(x.abs)
end
p sqrt(-4)
```

In the case of mix-in modules, you can refine concrete classes which
include mix-in modules.

class Foo
include BarMixin
end
module Baz
refine Foo do # instead of refine(BarMixin)
def method_in_bar_mixin
...
end
end
end

--
Shugo Maeda

=end

#18 - 11/25/2010 11:01 PM - shugo (Shugo Maeda)

=begin
2010/11/24 Shugo Maeda redmine@ruby-lang.org:

> As I said at RubyConf 2010, I'd like to propose a new features called
> "Refinements."

FYI, the slides of my talk are available at:

http://www.slideshare.net/ShugoMaeda/rc2010-refinements

or:

http://shugo.net/tmp/RubyConf2010.pdf

--
Shugo Maeda

=end

#19 - 11/25/2010 11:17 PM - ko1 (Koichi Sasada)

=begin
(2010/11/25 13:55), Shugo Maeda wrote:

> I have run "make benchmark" 5 times, and it shows that the modified
> version is slower than average 2.5% than the original version.

> > The benchmark result is available at:
> >
> > http://shugo.net/tmp/refinement-benchmark-20101107.csv

My benchmark result is here:
http://www.atdot.net/fp_store/f.132gcl/file.graph.png
(Y-axis is a speed-up ratio (up is better))

on ruby 1.9.3dev (2010-11-25 trunk 29925) [x86_64-linux]

raw data:
http://www.atdot.net/sp/view/402gcl

I want to note that the average score using YARV's benchmark is not good
idea :)

--
// SASADA Koichi at atdot dot net

=end

#20 - 11/26/2010 12:48 AM - mame (Yusuke Endoh)

=begin
Hi,

2010/11/25 SASADA Koichi ko1@atdot.net:

> (2010/11/25 13:55), Shugo Maeda wrote:
>
> > I have run "make benchmark" 5 times, and it shows that the modified
> > version is slower than average 2.5% than the original version.
> >
> > The benchmark result is available at:
> >
> > http://shugo.net/tmp/refinement-benchmark-20101107.csv
>
> My benchmark result is here:
> http://www.atdot.net/fp_store/f.132gcl/file.graph.png
> (Y-axis is a speed-up ratio (up is better))

I also conducted retest.
Up to 24.9% performance degradation occured on my environment.

(become slower)
24.9%: vm2_super
24.5%: loop_for
21.9%: vm1_ivar
20.9%: vm2_zsuper
20.7%: vm1_swap
15.7%: vm2_poly_method
15.0%: app_tak
12.7%: vm3_thread_mutex
11.6%: vm2_mutex
11.3%: so_nsieve
....
-1.6%: so_partial_sums
-2.6%: vm2_array
-3.0%: so_random
-4.0%: so_nbody
-4.3%: app_tarai
-5.0%: so_binary_trees
-5.1%: vm1_const
-7.1%: vm1_ensure
-7.5%: vm1_simplereturn
-7.8%: so_array
(become faster)

raw data:
http://www.atdot.net/sp/view/gu5gcl

csv:

```
program,original,refinement,time ratio (refinement / original)
app_answer,0.0971895694732666,0.10370779037475586,1.0670670827828104
app_erb,0.7143167495727539,0.724087905883789,1.0136790244900171
app_factorial,0.3495138168334961,0.35308108329772947,1.0102063675094504
app_fib,1.3584553718566894,1.383923053741455,1.0187475292986308
app_mandelbrot,0.31960186958312986,0.34449019432067873,1.0778729009627253
app_pentomino,31.527720880508422,33.666850090026855,1.0678491546415878
app_raise,0.8428834438323974,0.8716522693634033,1.0341314398111803
app_strconcat,0.5739905834197998,0.5915801525115967,1.0306443513184473
app_tak,1.676273727416992,1.9272963523864746,1.149750378392131
app_tarai,1.4925862312316895,1.427922534942627,0.9566767434028238
app_uri,1.4658241271972656,1.4655370235443115,0.9998041349929865
io_file_create,0.42700395584106443,0.44702463150024413,1.0468863938736708
io_file_read,0.39592618942260743,0.40063791275024413,1.0119005093714764
io_file_write,0.2036591053009033,0.20306005477905273,0.9970585625377982
loop_for,2.656430149078369,3.306622934341431,1.2447618603819273
loop_generator,0.8340113639831543,0.8517292499542236,1.0212441781206079
loop_times,2.4262061595916746,2.4653957366943358,1.0161526162761274
loop_whileloop,1.10532808303833,1.1343581676483154,1.026263771865985
loop_whileloop2,0.22827577590942383,0.23306760787963868,1.020991416855883
so_ackermann,1.4401548862457276,1.5601153373718262,1.083296909430914
so_array,2.876900815963745,2.6525750160217285,0.9220251881131089
so_binary_trees,0.7443643569946289,0.7073556423187256,0.950281452452498
so_concatenate,0.6735883235931397,0.6702359199523926,0.9950230674681766
so_count_words,0.33946728706359863,0.3388693332672119,0.99823855252281
so_exception,1.7248088836669921,1.778290367126465,1.0310071938786456
so_fannkuch,28.669377708435057,30.272033739089967,1.0559013190643263
so_fasta,3.9797119140625,3.990163469314575,1.0026262090014968
so_k_nucleotide,2.4159430503845214,2.584695339202881,1.0698494481446907
so_lists,0.5479694843292237,0.5965001106262207,1.0885644687977543
so_mandelbrot,9.189719247817994,9.212599325180054,1.00248974715604
so_matrix,0.7333712100982666,0.7263244152069092,0.9903912305332886
so_meteor_contest,9.425535678863525,9.882782745361329,1.0485115204139714
so_nbody,6.576598167419434,6.311960506439209,0.9597607069424975
so_nested_loop,2.17801456451416,2.1627822875976563,0.9930063475402418
so_nsieve,4.845090913772583,5.39297251701355,1.1130797363747225
so_nsieve_bits,5.009188795089722,5.168913412094116,1.0318863240213594
so_object,1.4976745128631592,1.5413443088531493,1.029158402319677
so_partial_sums,8.703983974456786,8.56481056213379,0.9840103781519562
so_pidigits,2.3049992084503175,2.294805479049683,0.9955775562250678
so_random,0.5043463706970215,0.48915743827819824,0.9698839264019453
so_reverse_complement,2.5664955139160157,2.577986478805542,1.0044772978667682
so_sieve,0.1414027690887451,0.15396690368652344,1.0888535258449785
so_spectralnorm,5.634414100646973,5.971523380279541,1.0598304053643943
vm1_block,4.044720840454102,4.3206627368927,1.06822272965756
vm1_const,2.2964776039123533,2.180025339126587,0.9492909207617028
vm1_ensure,1.3428500652313233,1.247154951095581,0.9287373053675523
vm1_ivar,2.4599918842315676,2.9994019985198976,1.2192731275846573
vm1_ivar_set,2.4699801445007323,2.4521598339080812,0.9927852413581838
vm1_length,2.1807806491851807,2.199760341644287,1.0087031643765723
vm1_neq,1.9046989917755126,1.9770766735076903,1.0379995380082125
vm1_not,1.5559280872344972,1.6081242561340332,1.0335466461000196
vm1_rescue,1.2895635128021241,1.297832202911377,1.0064120068745475
vm1_simplereturn,3.474906158447266,3.21527214050293,0.9252831569816115
vm1_swap,1.6546783447265625,1.9964598178863526,1.2065546299370165
vm2_array,1.432089138031006,1.3952114582061768,0.9742490332162336
vm2_case,0.46059327125549315,0.4548778057098389,0.9875910789359233
vm2_eval,28.152580356597902,29.735985946655273,1.0562437108783984
vm2_method,3.3265981674194336,3.469304418563843,1.0428985540069338
vm2_mutex,2.5152658462524413,2.8074283599853516,1.1161557193519764
vm2_poly_method,4.341252231597901,5.024772596359253,1.157447742793274
vm2_poly_method_ov,0.6244135856628418,0.6301047325134277,1.009114386652149
vm2_proc,1.251944398880005,1.3809085845947267,1.1030111128178643
vm2_regexp,2.2819092750549315,2.2643832683563234,0.9923195865452688
vm2_send,0.7383904933929444,0.8028009414672852,1.0872308739761414
vm2_super,1.1586995601654053,1.4471320629119873,1.2489277744313703
vm2_unif1,0.6489530563354492,0.6893436431884765,1.0622396126479587
vm2_zsuper,1.2690173149108888,1.5337296962738036,1.2085963510919495
vm3_gc,1.2378501892089844,1.2501022815704346,1.0098978797824312
vm3_thread_create_join,2.7109290599822997,2.7120433807373048,1.0004110475524624
vm3_thread_mutex,237.79514360427856,268.030443572998,1.1271485174610414
```

--
Yusuke Endoh mame@tsg.ne.jp

=end

#21 - 11/26/2010 02:02 AM - mame (Yusuke Endoh)

=begin
Hi,

2010/11/25 Shugo Maeda shugo@ruby-lang.org:

> How about to separate them into the following three patches?
>
> 1. changes of control frames and method lookup
> 2. Refinements support
> 3. nested methods support

Seems good.

> I forgot to mention that refinements are enabled in class or module
> definitions where Module#refine is called.
>
> ```
>  class Foo
>    refine Fixnum do
>      def /(other) quo(other) end
>    end
>
>    p 1 / 2  #=> (1/2)
>  end
> ```

Aha!

> Do we need Kernel#refine?

I want it.

> > I guess that this `using_refine' is useful itself (though its
> > name is arguable). In addition, it allows us to write
> > MyXmlFormat as follows:
> >
> > ```
> >  module MyXmlFormat
> >    using_refine(Integer) do
> >      def to_xml; ...; end
> >    end
> >
> >   using_refine(String) do
> >   def to_xml; ...; end
> >   end
> >
> >   using_refine(Hash) do
> >   def to_xml; ...; end
> >   end
> > end
> > ```
>
> I guess Module#refine works the same as using_refine in this case.

Hmm. Then, when no block is given to Module#refine, how about
adding an implicit block that includes the outer module?

```
module FooExt
refine(Foo)
...
end
```

(equivalent to)

```
module FooExt
refine(Foo) { include FooExt }
...
end
```

> I don't think we need new keywords even if it is a big change,

> because some essential features such as module inclusion
> have no keyword in Ruby.

Indeed. But conventionally, essential features that involve
code block (such as class/module definition, method definition
and control statements) have their special keywords.

> I guess it's unlikely that you need refine module functions in
> real-world applications.

I thought that one of motivating examples of this feature is
mathn.rb. In fact, mathn.rb changes Math module to complex-
aware one.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#22 - 11/26/2010 02:25 AM - mame (Yusuke Endoh)

=begin
Hi,

Are these intended?

(1)

```
class Foo
end
module FooExtCore
def foo
bar
end
def bar
end
end
module FooExt
refine(Foo) { include FooExtCore }
end
using(FooExt)
Foo.new.foo
#=> undefined local variable or method `bar'
```

I can guess what's going on, but this will be a FAQ...

(2)

```
class Foo
end
module FooExt
end
using FooExt
module FooExt
refine(Foo) { include FooExt }
def foo
end
end
Foo.new.foo
#=> undefined method `foo'
```

Note that I don't want to attack this proposal.
I'm really for this and I hope to help improve it.
Please don't misunderstand me :-)

--
Yusuke Endoh mame@tsg.ne.jp

=end

#23 - 11/26/2010 06:07 AM - judofyr (Magnus Holm)

=begin

Woah, this is very nice stuff! Some comments/questions:

1. Could you give an example of how it would behave if it had local rebinding as in classbox?

2. I don't like the idea of having both #using and #include. I don't want to think about which one to use; I just want to use the module! I think this is getting even more messy than with the new mixin-feature. Even right *now* people only use #include (and def included(mod); mod.extend ClassMethods; end) because That's How Modules Work™.

3. Is this expected behaviour?

module Ext
refine(Object) do
def to_json; "something"; end
end
end

Fixnum.send(:using, Ext)
"Hello".to_json # => Works because #using worked at the file scope

This makes it impossible to add refinements within a included-callback.

Other than that: This is definitely a contender to sliced bread :-)

// Magnus Holm

=end

#24 - 11/26/2010 07:35 AM - rkh (Konstantin Haase)

=begin
On Nov 25, 2010, at 18:02 , Yusuke ENDOH wrote:

> > Do we need Kernel#refine?
>
> I want it.

What would be the difference between Kernel#refine and Module#prepend?
I mean, what point do local refinements have in a global scope?

Konstantin

=end

#25 - 11/26/2010 07:39 AM - rkh (Konstantin Haase)

=begin

On Nov 25, 2010, at 22:06 , Magnus Holm wrote:

> 1. I don't like the idea of having both #using and #include. I don't want to think about which one to use; I just want to use the module! I think this is getting even more messy than with the new mixin-feature. Even right *now* people only use #include (and def included(mod); mod.extend ClassMethods; end) because That's How Modules Work™.

But what if you want to use refinements only in the module included, not the module it is included into?
This would be impossible. One could always do a def self.included(klass) klass.send(:using, self) end (in case your point #3 will be fixed).

Konstantin

=end

#26 - 11/26/2010 10:25 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/25 SASADA Koichi ko1@atdot.net:

> > I have run "make benchmark" 5 times, and it shows that the modified
> > version is slower than average 2.5% than the original version.
> >
> > The benchmark result is available at:
> >
> > http://shugo.net/tmp/refinement-benchmark-20101107.csv
>
> My benchmark result is here:
> http://www.atdot.net/fp_store/f.132gcl/file.graph.png
> (Y-axis is a speed-up ratio (up is better))
>
> on ruby 1.9.3dev (2010-11-25 trunk 29925) [x86_64-linux]
>
> raw data:
> http://www.atdot.net/sp/view/402gcl
>
> I want to note that the average score using YARV's benchmark is not good
> idea :)

I agree.

Please remember that I'm a daimyo programmer.
I'd like you to improve the performance:)

--
Shugo Maeda

=end

#27 - 11/26/2010 11:02 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Yusuke ENDOH mame@tsg.ne.jp:

> > How about to separate them into the following three patches?
> >
> > 1. changes of control frames and method lookup
> > 2. Refinements support
> > 3. nested methods support
>
> Seems good.

I'll make the patches.

> > Do we need Kernel#refine?
>
> I want it.

I agree with you. What do you think of it, Matz?

> > I guess Module#refine works the same as using_refine in this case.
>
> Hmm. Then, when no block is given to Module#refine, how about
> adding an implicit block that includes the outer module?
>
> ```
>  module FooExt
>    refine(Foo)
>    ...
>  end
> ```
>
> (equivalent to)
>
> ```
>  module FooExt
>    refine(Foo) { include FooExt }
>    ...
>  end
> ```

Could you tell me why you need this feature?

> > I don't think we need new keywords even if it is a big change,
> > because some essential features such as module inclusion
> > have no keyword in Ruby.
>
> Indeed. But conventionally, essential features that involve
> code block (such as class/module definition, method definition
> and control statements) have their special keywords.

I guess that most of these constructs have reasons why they need
keywords and special syntax. For example, class, module, and method
definitions take undefined identifiers (alias doesn't invoke code
block, but it's a keyword for the same reason), and some control
statements take expressions evaluated lazily. However, there is no
such reason for Refinements.

If refine is a keyword, there is one good thing. We don't need "do"
after class names.

```
refine Fixnum
...
end
```

I'm not sure it's an enough reason to introduce a new keyword.
What do you think of it, Matz?

> > I guess it's unlikely that you need refine module functions in
> > real-world applications.
>
> I thought that one of motivating examples of this feature is
> mathn.rb. In fact, mathn.rb changes Math module to complex-
> aware one.

If think it's better to provide complex-aware features as module functions
of a new module such as MathN or MathN::Math. Users can include it
instead of Math.

--
Shugo Maeda

=end

#28 - 11/26/2010 11:08 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Yusuke ENDOH mame@tsg.ne.jp:

> Are these intended?
>
> (1)
>
> ```
>  class Foo
>  end
>  module FooExtCore
>    def foo
>      bar
>    end
>    def bar
>    end
>  end
>  module FooExt
>    refine(Foo) { include FooExtCore }
>  end
>  using(FooExt)
>  Foo.new.foo
>    #=> undefined local variable or method `bar'
> ```

> I can guess what's going on, but this will be a FAQ...

This behavior is intended, and you need define foo and bar inside the
block passed to refine.
I don't know any way to make the above code work without local rebinding.

> (2)
>
> ```
>  class Foo
>  end
>  module FooExt
>  end
>  using FooExt
>  module FooExt
>    refine(Foo) { include FooExt }
>    def foo
>    end
>  end
>  Foo.new.foo
>    #=> undefined method `foo'
> ```

I'd like to allow the above code if possible, but it's hard for me.

> Note that I don't want to attack this proposal.
> I'm really for this and I hope to help improve it.
> Please don't misunderstand me :-)

I know it. Thanks for your helpful comments.

--
Shugo Maeda

=end

#29 - 11/26/2010 11:24 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Magnus Holm judofyr@gmail.com:

> Woah, this is very nice stuff! Some comments/questions:
>
> 1. Could you give an example of how it would behave if it had local rebinding as in classbox?

If it had local rebinding, the following code would print "Quux::Foo#bar".

class Foo
def bar
puts "Foo#bar"
end

def baz
bar
end
end

module Quux
refine Foo do
def bar
puts "Quux::Foo#bar"
end
end
end

using Quux
foo = Foo.new
foo.baz

> 1. I don't like the idea of having both #using and #include. I don't want to think about which one to use; I just want to use the module! I think this is getting even more messy than with the new mixin-feature. Even right *now* people only use #include (and def included(mod);

mod.extend ClassMethods; end) because That's How Modules Work™.

Matz doesn't like it, but I think it's worth considering.
However, it's a problem that we have main#include, where main is self
at the top-level, but not Kernel#include now.

1. Is this expected behaviour?

```
module Ext
 refine(Object) do
   def to_json; "something"; end
 end
end

Fixnum.send(:using, Ext)
"Hello".to_json # => Works because #using worked at the file scope
```

It is intended.

This makes it impossible to add refinements within a included-callback.

If a binding is passed to include, it may be possible as follows:

```
module Foo
refine XXX do ... end
def do_something
end
...
def self.included(klass, binding)
eval("using Foo", binding)
end
end

class Bar
include Foo
# both do_something and refinements are available
end
```

It's necessary to check the arity of included for backward compatibility.

Currenty, you can use a used-callback instead.

```
module Foo
refine XXX do ... end
def do_something
end
...
def self.used(klass)
klass.send(:include, self)
end
end

class Bar
using Foo
end
```

--
Shugo Maeda

=end

#30 - 11/26/2010 11:26 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Haase, Konstantin Konstantin.Haase@student.hpi.uni-potsdam.de:

Do we need Kernel#refine?

I want it.

> What would be the difference between Kernel#refine and Module#prepend?
> I mean, what point do local refinements have in a global scope?

Kernel#refine doesn't refine classes in a global scope, but in a file
or method scope.
If you call Kernel#refine at the top-level, the refinements are
enabled only in that file.
If you call Kernel#refine in a method, the refinements are enabled
only in the method.

--
Shugo Maeda

=end

#31 - 11/26/2010 11:29 AM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Jos Backus jos@catnook.com:

> Regarding naming: how about using the name use' instead ofusing'? This
> would be in line with other names such as include',extend', etc.

Some other people said the same, but I think using' is better than
use' because `use' is already used in Rack.

> Thanks for working on this, it looks like a cool and useful feature.

Thank you.

--
Shugo Maeda

=end

#32 - 11/26/2010 12:45 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/11/26 Shugo Maeda shugo@ruby-lang.org:

> > > I guess Module#refine works the same as using_refine in this case.
> >
> > Hmm. Then, when no block is given to Module#refine, how about
> > adding an implicit block that includes the outer module?
> >
> > ```
> >  module FooExt
> >    refine(Foo)
> >    ...
> >  end
> > ```
> >
> > (equivalent to)
> >
> > ```
> >  module FooExt
> >    refine(Foo) { include FooExt }
> >    ...
> >  end
> > ```
>
> Could you tell me why you need this feature?

Because it requires less indentation, I thought.
But I found out that it has a problem of ruby-core:33386...

> > > I don't think we need new keywords even if it is a big change,
> > > because some essential features such as module inclusion

have no keyword in Ruby.

Indeed. But conventionally, essential features that involve
code block (such as class/module definition, method definition
and control statements) have their special keywords.

I guess that most of these constructs have reasons why they need
keywords and special syntax.

I don't think so. "class Foo; end" can be written as "Foo =
Class.new { }" (though there are indeed subtle differences between
them).

If refine is a keyword, there is one good thing. We don't need "do"
after class names.

```
refine Fixnum
  ...
end
```

The API design that "def" statements are put in a Ruby's block,
is slightly weird (for me). I guess that there is no precedent of
such a style in Ruby's embedded featues, except meta programming
(such as Class.new and class_eval).
From now on, does Ruby encourage such a style in casual use?

--
Yusuke Endoh mame@tsg.ne.jp

=end

#33 - 11/26/2010 01:39 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/26 Yusuke ENDOH mame@tsg.ne.jp:

```
module FooExt
  refine(Foo)
  ...
end
```

(equivalent to)

```
module FooExt
  refine(Foo) { include FooExt }
  ...
end
```

Could you tell me why you need this feature?

Because it requires less indentation, I thought.

I see. refine without blocks looks confusing for me because it works
different from refine with a block.

But I found out that it has a problem of ruby-core:33386...

I think the above code should work because the refinement of Foo is
enabled in FooExt.
It may be a bug.

I don't think we need new keywords even if it is a big change,
because some essential features such as module inclusion
have no keyword in Ruby.

Indeed. But conventionally, essential features that involve
code block (such as class/module definition, method definition
and control statements) have their special keywords.

I guess that most of these constructs have reasons why they need
keywords and special syntax.

I don't think so. "class Foo; end" can be written as "Foo =
Class.new { }" (though there are indeed subtle differences between
them).

"refine Foo do end" is different from "Foo = Class.new {}" because
"refine Foo do end" looks good, but "Foo = Class.new {}" doesn't.
I think how it looks is more important than whether it uses keywords or not.

If refine is a keyword, there is one good thing. We don't need "do"
after class names.

```
 refine Fixnum
   ...
 end
```

The API design that "def" statements are put in a Ruby's block,
is slightly weird (for me). I guess that there is no precedent of
such a style in Ruby's embedded featues, except meta programming
(such as Class.new and class_eval).
From now on, does Ruby encourage such a style in casual use?

I think Module#refine is a meta programming feature like class_eval,
and most application programmers need not use it directly.
And, "refine Foo do end" looks not so bad, so I think the new keyword
refine has more cons than pros.

--
Shugo Maeda

=end

#34 - 11/27/2010 01:39 PM - shugo (Shugo Maeda)

- File control_frame_change-r29944-20101127.diff added

- File refinements-r29944-20101127.diff added

- File nested_methods-r29944-20101127.diff added

=begin
Hi,

How about to separate them into the following three patches?

1. changes of control frames and method lookup
2. Refinements support
3. nested methods support

I have attached these three patches.
Please check them.

The following code works now:

```
class Foo; end

module FooExt
refine(Foo) { include FooExt }
def foo
puts "foo"
end
def bar
```

puts "bar"
foo
end
end

using FooExt
f = Foo.new
f.foo
f.bar

=end

#35 - 11/27/2010 01:57 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33393] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Fri, 26 Nov 2010 11:02:28 +0900, Shugo Maeda shugo@ruby-lang.org writes:

|>> Do we need Kernel#refine?
|>
|> I want it.
|
|I agree with you. What do you think of it, Matz?

I don't see strong requirement, but I accept.

```
                            matz.
```

=end

#36 - 11/27/2010 02:10 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33396] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Fri, 26 Nov 2010 11:24:43 +0900, Shugo Maeda shugo@ruby-lang.org writes:

|Matz doesn't like it, but I think it's worth considering.
|However, it's a problem that we have main#include, where main is self
|at the top-level, but not Kernel#include now.

If we introduce local rebinding, I think we have to rename it to
"classbox" from "refinement". Besides that, we sill have lot of
issues about implementation, performance, etc. But I believe, Shugo,
you are the key person. I am the one who approve.

```
                            matz.
```

=end

#37 - 11/27/2010 05:35 PM - Chauk-Mean (Chauk-Mean Proum)

=begin

> > Regarding naming: how about using the name use' instead ofusing'? This
> > would be in line with other names such as include',extend', etc.
>
> Some other people said the same, but I think using' is better than
> use' because `use' is already used in Rack.

IMHO, it would be a shame to have such a new core feature with a "strange" naming.
I also prefer use instead of using.
May be the core team can request Rack developers to rename their use method to e.g. rack_use.
I guess that this new feature will be available only for ruby-1.9.3+.
So this leaves time for Rack developers and users to migrate their code base.

There are already some "strange" namings for some methods (http://redmine.ruby-lang.org/issues/show/4065),
so please take care of consistent naming for new methods.

Having said that, the feature looks very interesting.

=end

#38 - 11/27/2010 05:40 PM - rkh (Konstantin Haase)

=begin

On Nov 27, 2010, at 09:35 , Chauk-Mean Proum wrote:

> Issue #4085 has been updated by Chauk-Mean Proum.
>
> > > Regarding naming: how about using the name use' instead ofusing'? This
> > > would be in line with other names such as include',extend', etc.
> >
> > Some other people said the same, but I think using' is better than
> > use' because `use' is already used in Rack.
>
> IMHO, it would be a shame to have such a new core feature with a "strange" naming.
> I also prefer use instead of using.
> May be the core team can request Rack developers to rename their use method to e.g. rack_use.
> I guess that this new feature will be available only for ruby-1.9.3+.
> So this leaves time for Rack developers and users to migrate their code base.

This would be about any Ruby web application/framework/library out there. Not only Rack implements use, but Rails and Sinatra (and probably others) do so, too, in order to behave like Rack.

Konstantin

=end

#39 - 11/27/2010 05:43 PM - rkh (Konstantin Haase)

=begin

On Nov 27, 2010, at 06:10 , Yukihiro Matsumoto wrote:

> Hi,
>
> In message "Re: [ruby-core:33396] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
> on Fri, 26 Nov 2010 11:24:43 +0900, Shugo Maeda shugo@ruby-lang.org writes:
>
> |Matz doesn't like it, but I think it's worth considering.
> |However, it's a problem that we have main#include, where main is self
> |at the top-level, but not Kernel#include now.
>
> If we introduce local rebinding, I think we have to rename it to
> "classbox" from "refinement". Besides that, we sill have lot of
> issues about implementation, performance, etc. But I believe, Shugo,
> you are the key person. I am the one who approve.
>
> ```
>                           matz.
> ```

So, is local rebinding still on the table? From my point of view local rebinding is far more intuitive and there are some use cases I am not sure I could solve without local rebinding.

Konstantin

=end

#40 - 11/27/2010 05:46 PM - lsegal (Loren Segal)

=begin

On 11/27/2010 3:35 AM, Chauk-Mean Proum wrote:

> IMHO, it would be a shame to have such a new core feature with a "strange" naming.
> I also prefer use instead of using.
> May be the core team can request Rack developers to rename their use method to e.g. rack_use.
> I guess that this new feature will be available only for ruby-1.9.3+.
> So this leaves time for Rack developers and users to migrate their code base.
> This is really bad release engineering. You don't just change a method
> name because it's "strange". "Migrating" a codebase doesn't work when

> you need to support older Ruby versions, it only makes things messier
> for no good reason. FWIW, since you brought it up, your linked issue has
> the same problem of completely breaking backwards compat. for nothing
> but vanity, and I wouldn't be fond of that either.

Personally I see no problem with using. I'm not sure what the fuss is
about.

- Loren

=end

#41 - 11/27/2010 10:32 PM - Chauk-Mean (Chauk-Mean Proum)

=begin
11/27/2010 09:46 AM - Loren Segal wrote :

> > I also prefer use instead of using.
> > May be the core team can request Rack developers to rename their use method to e.g. rack_use.
> > I guess that this new feature will be available only for ruby-1.9.3+.
> > So this leaves time for Rack developers and users to migrate their code base.
> > This is really bad release engineering. You don't just change a method name because it's "strange".

Yes, I know that changes that break backward compatibility should be avoided.

> "Migrating" a codebase doesn't work when you need to support older Ruby versions,

In this case, the impact is on Rack and Web frameworks that rely on Rack (and I agree that there is lot as pointed by Konstantin). But there is no impact on older Ruby versions as refine/using is a new feature.
It is also just a proposal.

> FWIW, since you brought it up, your linked issue has the same problem of completely
> breaking backwards compat. for nothing but vanity, and I wouldn't be fond of that either.

Vanity ?
It seems that I was not alone to prefer use to using.
Regarding the proposal for renaming append_features, it was only to make the relationship with include more clear.
If other people think that this is not worth breaking the compatibility, then that's fine.

Please leave other people express their opinion, a different opinion does not mean vanity.

Chauk-Mean.

=end

#42 - 11/27/2010 11:27 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33421] [Ruby 1.9-Feature#4085] Refinements and nested methods"
on Sat, 27 Nov 2010 17:35:43 +0900, Chauk-Mean Proum redmine@ruby-lang.org writes:

|>Some other people said the same, but I think using' is better than
|>use' because `use' is already used in Rack.
|
|IMHO, it would be a shame to have such a new core feature with a "strange" naming.
|I also prefer use instead of using.

For specifying namespace, "using" is used in many languages, C++, C#
etc. On the other hand, as far as I know, no language use the keyword
"use" for namespaces, but for other purposes. For example, Perl6's
"use" is to mix-in traits, just like Ruby's "include".

So, from language designers' view, "using" is not that strange.

```
                            matz.
```

=end

#43 - 11/27/2010 11:46 PM - Chauk-Mean (Chauk-Mean Proum)

=begin
Hi Matz,

> For specifying namespace, "using" is used in many languages, C++, C#
> etc. On the other hand, as far as I know, no language use the keyword
> "use" for namespaces, but for other purposes. For example, Perl6's
> "use" is to mix-in traits, just like Ruby's "include".
>
> So, from language designers' view, "using" is not that strange.

'using' is not strange in itself.
It's just that it is not "in line with other names such as include',extend', etc." as other people said.
You're Ruby's grand master, the decision is definitively yours.

Chauk-Mean.

=end

#44 - 11/28/2010 01:10 AM - judofyr (Magnus Holm)

=begin
Thanks for your answers,

On Fri, Nov 26, 2010 at 03:24, Shugo Maeda shugo@ruby-lang.org wrote:

> Hi,
>
> 2010/11/26 Magnus Holm judofyr@gmail.com:
>
>> Woah, this is very nice stuff! Some comments/questions:
>>
>> 1. Could you give an example of how it would behave if it had local rebinding as in classbox?
>
> If it had local rebinding, the following code would print "Quux::Foo#bar".
>
> class Foo
> def bar
>   puts "Foo#bar"
> end
>
> def baz
>   bar
> end
> end
>
> module Quux
> refine Foo do
>   def bar
>   puts "Quux::Foo#bar"
>   end
> end
> end
>
> using Quux
> foo = Foo.new
> foo.baz

Thanks a lot. Could you explain why included modules are rebound though?

```
class CharArray
  include Enumerable

  def initialize(str)
    @array = str.unpack("C*")  # Unpacks to integers
  end

  def each(&blk)
    @array.each(&blk)
  end
  def map
    res = []
    each { |val| res << yield(val) }
    res
  end
end
```

```
test = CharArray.new("Hello World")
test.each { |x| p x }  # Prints a list of integers (expected)
```

```
module CharArrayStr
  refine CharArray do
    def each
      super { |c| yield c.chr }
    end
  end
end
```

```
using CharArrayStr
test.each   { |x| p x }  # Prints a list of strings (expected)
test.map    { |x| p x }  # Prints a list of integers (exptected;
```

no local rebinding)
test.select { |x| p x } # Prints a list of strings?!

=end

#45 - 11/30/2010 12:29 PM - mame (Yusuke Endoh)

=begin
Hi,

Sorry for late reply.

2010/11/26 Shugo Maeda shugo@ruby-lang.org:

> > > Because it requires less indentation, I thought.
>
> I see. refine without blocks looks confusing for me because it works
> different from refine with a block.

Maybe another name is needed?

I think that the short name `refine' is more appropriate to the non-
block style than the block style because I believe the non-block
style is more suitable for casual use. It requires less indentation,
and it complies with traditional style (like Module#include).

A longer (more "meta-programming-like") name would be appropriate to
the block style, such as Module#refine_class, #refine_class_eval,
#class_eval_with_refinement.

> > > > I guess that most of these constructs have reasons why they need
> > > > keywords and special syntax.
> > >
> > > I don't think so. "class Foo; end" can be written as "Foo =
> > > Class.new { }" (though there are indeed subtle differences between
> > > them).
>
> "refine Foo do end" is different from "Foo = Class.new {}" because
> "refine Foo do end" looks good, but "Foo = Class.new {}" doesn't.
> I think how it looks is more important than whether it uses keywords or not.

"refine Foo do def ... end end" looks not so good to me.

> > > > If refine is a keyword, there is one good thing. We don't need "do"
> > > > after class names.
> > > >
> > > > refine Fixnum
> > > > ...

end

> > The API design that "def" statements are put in a Ruby's block,
> > is slightly weird (for me). I guess that there is no precedent of
> > such a style in Ruby's embedded featues, except meta programming
> > (such as Class.new and class_eval).
> > From now on, does Ruby encourage such a style in casual use?
>
> I think Module#refine is a meta programming feature like class_eval,
> and most application programmers need not use it directly.

I guess you think so because we are not used to the feature yet.
If it is really just a meta-programming feature, the name should be
more "meta-programming-like".

The non-block style has a precedent (Module#include), so, if it is
adopted, I agree that any new keyword is not needed. Otherwise, I
prefer a new keyword to a new weird (to me) coding style.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#46 - 11/30/2010 07:20 PM - headius (Charles Nutter)

=begin
This is a long response, and for that I apologize. I want to make sure
I'm being clear about my concerns, so they can be addressed in a
meaningful way.

SUMMARY:

- "using" not being a keyword requires all calls to check for refinements all the time, globally degrading performance.
- instance_eval propagating refinements requires all block invocations to localize what refinements they use for invocation on every activation.
- shared, mutable structures can't be used to store the active refinement, due to concurrency issues
- there are very likely many more complexities than illustrated here that result from the combination of runtime-mutable lexical scoping structures, concurrency, and method caching.

On Wed, Nov 24, 2010 at 7:12 AM, Shugo Maeda redmine@ruby-lang.org wrote:

> If a module or class is using refinements, they are enabled in
> module_eval, class_eval, and instance_eval if the receiver is the
> class or module, or an instance of the class.

I am surprised nobody else has questioned this behavior. I believe it
is a problem.

Currently, blocks handle method dispatch like any other scope...i.e.
they look only at the "self" object's class (for fcall/vcall) or the
target object's class (for normal call). A typical caching structure
to optimize this then has an entry that stores previously-seen
method(s) and invalidates based on some trivial guard. In 1.9, this is
a global serial number. In JRuby, it's a class hierarchy-based guard.

The global serial number approach in 1.9 means that any change that
flip that serial number cause all caches everywhere to invalidate.
Normally this only happens on method definition or module inclusion,
which is why defining methods or including modules at runtime is
strongly discouraged for performance reasons.

Refinements make method lookup more complicated, since now any scope
where a refinement can no longer use the simple "target class" lookup
and cache-validation logic. Because refinements are enabled at
runtime, after parse, this also means that all calls everywhere must
constantly check if a refinement is enabled. This is performance hit
#1.

If "using" were a keyword, we could know at parse time that some calls
must check for refinements and other calls do not need to, localizing
the performance hit to only scopes where refinements are active. I

would strongly encourage "using" be made into a keyword.

Without "using" being a keyword, we can still avoid a global performance hit by pretending it's a keyword and proactively changing how scopes are parsed in the presence of "using" in a containing scope. This is likely what we would do in JRuby, forcing all class-body calls named "using" to "damage" performance in child scopes. We would also disallow or strongly discourage aliasing of "using", since it would be impossible at parse time to make a proper decision. We already do this for methods like "eval", which force a method body to be completely deoptimized.

The instance_eval case basically makes it impossible to avoid the performance hit for method calls within a block, since at any time a previously-captured block could be instance_eval'ed against a receiver class with refinements enabled. So all blocks everywhere would have to constantly check for refinements, forever. One possible suggestion to get around this would be to make all method calls in blocks check a global serial number, as in CRuby. At best, this is still an additional check in implementations that don't use a global serial number to invalidate method caches. At worst, it's still infeasible.

Recall that previously, refinements were largely lexical and morely static. In other words, even though refinements would not be applied at parse time, they would be applied at method-definition time and unchangeable from then on. The instance_eval case throws this out completely. The same block could be instance_eval'ed against two different refinements at the same time. Since the current logic stores the active refinement in the cache, and the cache is shared across all invocations (including concurrent invocations), we now have a case where mid-call, the static in-memory code/caches for a block would have to switch to a different refinement. Obviously this is intractable, since we wanted refinements in the first place for their isolation characteristics. In order to avoid this, all blocks everywhere would need to *never* cache method lookups, and always do a full slow-path lookup on their thread-local structures.

Even if an implementation isn't actually concurrent, things are still intractable, since any invocation of instance_eval against a refinement would have to force a global serial number change (at minimum) to force caches to be invalidated. This means that any use anywhere of instance_eval against a refinement would cause all block-borne method calls to flush and recache their next invocation.

And if that's not bad enough, even on a non-concurrent implementation the context-switch boundaries are finer-grained than individual calls...so any shared mutable data structures indicating what refinement to use would *still* require slow-path lookup every time in order to isolate one refinement-targeted instance_eval's effects from others.

And even if we don't consider concurrency, there's the issue of the *same* block being used in the *same* call stack for *different* refinements. Any call you make downstream from a given block could cause that block's static in-memory structures to be modified.

It might be possible to reduce the slow-path logic to checking the call frame for *every single call* to see if a refinement is active, and then if the caller knows that a refinement is active call frames would have to have this bit set. But the caller is not responsible for the call frame construction, so all calls everywhere would have to check the caller's frames to see if a refinement is active. Accessing the caller's frame means every call needs to do additional pointer dereferences and checks for every Ruby method call.

And one last case that's a problem: author's intent. If I write a block of code that does "1 + 1", I intend for that code to do normal Fixnum#+, and I intend for the result to be 2. It should not be possible for a caller to change the intent of my code just because I passed it in a bock. This has been my argument against using blocks as bindings, and it's another argument against instance_eval being able to force refinments into "someone else's code".

I could continue to try to theorize about possible implementations, but they all lead toward runtime-alterable refinements being a

devlishly complicated thing to implement and potentially impossible to
optimize. I could be wrong, especially if my understanding about the
feature is flawed.

Now, some positive reinforcement for "using" being a keyword and
instance_eval not propagating refinements.

If "using" were a keyword, calls within the related lexical scopes
would become "refinement-aware calls", localizing performance impact
to only those calls. They would need additional cache guards,
potentially with global impact, but at least normal code would work
exactly as it does today. Block bodies would be no exception; unless a
"using" were active at parse time, all calls could be taken at face
value. This would also preclude instance_eval of a block propagating
refinements, since the parse-time nature of "using" would mean a block
is what it is, forever. Your code can't modify the intent of my code,
and only calls where a parent scope at parse time contains the "using"
keyword would know anything about refinements.

This is, in fact, how I implemented "selector namespaces" over a year
ago in JRuby, as an example.
http://groups.google.com/group/ruby-core-google/msg/6f45dcb363e75267

I can try to come up with a concrete example of the problems with the
current proposal and implementation, but the concurrency cases would
be difficult to show.

- Charlie

=end

#47 - 11/30/2010 08:09 PM - jedediah (Jedediah Smith)

=begin
It would be great to finally have scoped monkey patching for Ruby, but I am finding some problems with this approach.

There are many ways for refinements to leak out of the lexical scope that uses them:

1. Inheritance (which is used in many APIs e.g. ActiveRecord::Base)
2. DSLs that use instance_eval
3. Re-opening a class
4. When a class that uses refinements is itself refined (or does it not work that way?)

Classes that are to be used in any of these ways can't use refinements internally, I guess. For 1 and 4, that could be any class at all. So we still have to be paranoid about using refinements in library code, which kind of defeats the purpose.

Also, if "using" and "include" both operate on modules, the difference between them is going to be confusing, particularly for those unfamiliar with the history of the language. It seems arbitrary to overload modules with a use unrelated to their existing purpose. And having to group refinements in modules seems overly complicated for many cases. I would like to be able to do something like this:

refinement StringExt < String
def pig_latin
"#{self[1..-1]}#{self[0]}ay"
end
end

...

using StringExt
"woot".pig_latin   # ==> "ootway"

But there does need to be a way to group refinements, and there should probably be a way to combine groups as well. Perhaps this:

refinement CoreExt   # empty refinement bound to constant CoreExt
using ArrayExt   # import other refinement(s)

```
refinement < String        # anonymous refinement of String
  ...
end
```

end

This keeps refinements and modules separate and makes their usage clear: modules are "included" dynamically, refinements are "used" lexically. Mix them up and you get a TypeError. It also keeps definition of refinements separate from their use, and lets you use an anonymous refinement inside a module without exporting it. Refinements can still be nested members of modules. The above syntax may not be quite right, but I think the semantics are: refinements are composable sets of lexically scoped monkey patches.

=end

#48 - 12/02/2010 08:30 PM - shugo (Shugo Maeda)

=begin
Sorry for the delay. I had acute gastroenteritis....

2010/11/27 Haase, Konstantin Konstantin.Haase@student.hpi.uni-potsdam.de:

> So, is local rebinding still on the table? From my point of view local rebinding is far more intuitive and there are some use cases I am not sure I could solve without local rebinding.

Could you tell me one of the use cases?

--
Shugo Maeda

=end

#49 - 12/02/2010 08:41 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/27 Yukihiro Matsumoto matz@ruby-lang.org:

> |Matz doesn't like it, but I think it's worth considering.
> |However, it's a problem that we have main#include, where main is self
> |at the top-level, but not Kernel#include now.
>
> If we introduce local rebinding, I think we have to rename it to
> "classbox" from "refinement". Besides that, we sill have lot of
> issues about implementation, performance, etc. But I believe, Shugo,
> you are the key person. I am the one who approve.

Thanks for your encouragement.

I guess it's hard to get impeccable conclusion.
I'll express my preferences, but I would like you to make the final decision.

--
Shugo Maeda

=end

#50 - 12/02/2010 08:47 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/28 Magnus Holm judofyr@gmail.com:

> Thanks a lot. Could you explain why included modules are rebound though?

It was a bug. Please try the following three patches instead of
refinement-r29837-20101124.diff.

control_frame_change-r29944-20101127.diff
refinements-r29944-20101127.diff
nested_methods-r29944-20101127.diff

They are available at:

http://redmine.ruby-lang.org/issues/show/4085

--
Shugo Maeda

=end

#51 - 12/02/2010 09:34 PM - rkh (Konstantin Haase)

=begin
On Dec 2, 2010, at 12:30 , Shugo Maeda wrote:

> 2010/11/27 Haase, Konstantin Konstantin.Haase@student.hpi.uni-potsdam.de:
>
>> So, is local rebinding still on the table? From my point of view local rebinding is far more intuitive and there are some use cases I am not sure I could solve without local rebinding.
>
> Could you tell me one of the use cases?

When writing an asynchronous Rack application, you cannot use Rack::Lint, since you return a status code of -1 to signal your Rack handler that you will respond to the incoming request later. One option would be to completely disable Rack::Lint, which might not be what you want and is a bit painful, as it is hardcoded to be used in development mode in Rack. One could monkey-patch Rack::Lint directly, but it would be even better if it only excepts -1 for your application:

```
module AsyncRack
    refine Rack::Lint do
        def check_status(status)
            super unless status == -1
        end
    end
end

using AsyncRack
```

The complete async rack library (https://github.com/rkh/async-rack) could be implemented that way. Evil hacks for replacing constants are necessary at the moment, though it would also be solvable - modulo having the changes only locally instead of globally - by the proposed Module#prepend.

I think in general there are two rather distinct use case groups: Those where I don't know and don't want to have to care about the internals of the class that's being refined and those where I do and explicitly want to reach in deep to change a single internal (say in Rails I want to change how class names are mapped to files only in one initializer). If I don't have local rebinding, I would still have to care about the original implementation in order to figure out what methods are calling the method I want to change. In the Enumerable example I would have to override about every method, not only each, in order to change the behavior consistently.

Konstantin

=end

#52 - 12/02/2010 09:55 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/30 Yusuke ENDOH mame@tsg.ne.jp:

>>> Because it requires less indentation, I thought.
>>
>> I see. refine without blocks looks confusing for me because it works
>> different from refine with a block.
>
> Maybe another name is needed?
>
> I think that the short name `refine' is more appropriate to the non-
> block style than the block style because I believe the non-block
> style is more suitable for casual use. It requires less indentation,
> and it complies with traditional style (like Module#include).

It doesn't make sense because Module#include is a very different
feature from refine.

My proposal is to use modules as namespaces for refinements. So
indentation is a necessary evil. Otherwise, we need syntax like Java
packages and one file for each package.

> A longer (more "meta-programming-like") name would be appropriate to
> the block style, such as Module#refine_class, #refine_class_eval,
> #class_eval_with_refinement.

Meta-programming means programming on programs, so the non-block style
is also a meta-programming feature. Why should only the block style
be named more "meta-programming-like"?

I guess that most of these constructs have reasons why they need

keywords and special syntax.

I don't think so. "class Foo; end" can be written as "Foo =
Class.new { }" (though there are indeed subtle differences between
them).

"refine Foo do end" is different from "Foo = Class.new {}" because
"refine Foo do end" looks good, but "Foo = Class.new {}" doesn't.
I think how it looks is more important than whether it uses keywords or not.

"refine Foo do def ... end end" looks not so good to me.

Could you tell me why

```
refine Foo
def bar; end
end
```

is good but

```
refine Foo do
def bar; end
end
```

is not so good?

They look similar for me. The latter has "do", but it seems a good
word in this context.
I'm not sure because I'm not a good English writer.

The API design that "def" statements are put in a Ruby's block,
is slightly weird (for me). I guess that there is no precedent of
such a style in Ruby's embedded featues, except meta programming
(such as Class.new and class_eval).
From now on, does Ruby encourage such a style in casual use?

I think Module#refine is a meta programming feature like class_eval,
and most application programmers need not use it directly.

I guess you think so because we are not used to the feature yet.
If it is really just a meta-programming feature, the name should be
more "meta-programming-like".

I don't know why meta-programming features should have long names.
In Ruby, meta-programming is encouraged, and meta-programming features
sometimes have a short name such as eval, but rarely have a keyword.

The non-block style has a precedent (Module#include), so, if it is
adopted, I agree that any new keyword is not needed. Otherwise, I
prefer a new keyword to a new weird (to me) coding style.

Are precedents so important for innovations?

--
Shugo Maeda

=end

#53 - 12/02/2010 11:20 PM - lsegal (Loren Segal)

=begin

On 12/2/2010 7:55 AM, Shugo Maeda wrote:

> Could you tell me why
>
> refine Foo
> def bar; end
> end
>
> is good but
>
> refine Foo do
> def bar; end
> end
>
> is not so good?

The "refine do ... end" block implies a method call which therefore
becomes easily overridable at run time. This means the compiler cannot
statically compute refinements unless it just assumed "refine" was never
overridden. If refinements are truly meant to be lexically scoped, this
should be reflected in the compiler's handling of them. Charles Nutter's
post illustrates why this might matter.

- Loren

=end

#54 - 12/02/2010 11:39 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/12/2 Shugo Maeda shugo@ruby-lang.org:

> My proposal is to use modules as namespaces for refinements. So
> indentation is a necessary evil. Otherwise, we need syntax like Java
> packages and one file for each package.

I'm not against using modules as namespaces and refinement scope,
but I don't like to see the same module being used for refinement
and traditional use at the same time.
Modules for mix-in, modules for collection of helper methods, and
modules for refinement should be separated, at least, in casual
use.

We can clearly see that the following module FooExt is only for
refinement. Thus I like this style.

module FooExt
refine Foo
def ... end
end

>> A longer (more "meta-programming-like") name would be appropriate to
>> the block style, such as Module#refine_class, #refine_class_eval,
>> #class_eval_with_refinement.
>
> Meta-programming means programming on programs, so the non-block style
> is also a meta-programming feature. Why should only the block style
> be named more "meta-programming-like"?

I can call it "not for casual use" instead of "meta-programming-
like."

> Could you tell me why
>
> refine Foo
> def bar; end
> end

> is good but
>
> refine Foo do
> def bar; end
> end
>
> is not so good?

Because block includes method definition. A block looks to me
"dynamic" behavior, while method definition (using `def' keyword)
looks "static" behavior.
Of course I know that both are also evaluated dynamically in
Ruby, but I don't think that Ruby encourages such a style so much.

> I don't know why meta-programming features should have long names.
> In Ruby, meta-programming is encouraged, and meta-programming features
> sometimes have a short name such as eval, but rarely have a keyword.

Ah, we found the perception gap between I and you.
I do NOT think that meta-programming is so encouraged even in Ruby.
It is like "a trick," and should be used only when it is appropriate.
If it was really so encouraged, Ruby would not provide many
syntax, such as def' andclass.'

And I guess that "eval" just came from Perl.

>> The non-block style has a precedent (Module#include), so, if it is
>> adopted, I agree that any new keyword is not needed. Otherwise, I
>> prefer a new keyword to a new weird (to me) coding style.
>
> Are precedents so important for innovations?

I like this feature as new OO paradigm, but syntax is another
matter.
I also like traditional syntax design principle --including
"syntax-like convention" such as Kernel#require and #include--
and I want this feature also to respect it.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#55 - 12/03/2010 12:43 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/11/30 Charles Oliver Nutter headius@headius.com:

> The global serial number approach in 1.9 means that any change that
> flip that serial number cause all caches everywhere to invalidate.
> Normally this only happens on method definition or module inclusion,
> which is why defining methods or including modules at runtime is
> strongly discouraged for performance reasons.

Sorry I'm not sure that I could follow your argument about performance,
so I may miss your point.

I guess that casual users will execute all refinements immediately after
program is started, like class definition and method definition.
Thus, the global serial number approach will work well for refinement
in main use cases, I think.
In the sense, nested function by using refinements may be a problem.

> And one last case that's a problem: author's intent. If I write a
> block of code that does "1 + 1", I intend for that code to do normal
> Fixnum#+, and I intend for the result to be 2. It should not be
> possible for a caller to change the intent of my code just because I
> passed it in a bock. This has been my argument against using blocks as

> bindings, and it's another argument against instance_eval being able
> to force refinments into "someone else's code".

This is not a problem, but rather improvement. There is already open
class which so often breaks your intent. Refinements may also break
your intent, but it is less often and more controllable than open class.

> Now, some positive reinforcement for "using" being a keyword and
> instance_eval not propagating refinements.

I'm not against your proposal, but I wonder if it does not make sense
because we can still write: eval("using FooExt")
To address your concern, `using' keyword should have a block:

```
using FooExt
# FooExt enabled
end
# FooExt disabled
```

I don't like this syntax because of more indentation, though.

> I can try to come up with a concrete example of the problems with the
> current proposal and implementation, but the concurrency cases would
> be difficult to show.

I might find serious concurrency problem of Shugo's patch, though I'm
not sure that this is what you mean. Indeed, we may have to give up
propagating refinements via block.

```
class C
def test; p :test; end
end
module FooExt
refine(C) { def test; p :foo; end }
end
module BarExt
refine(C) { def test; p :bar; end }
end
f = proc do
sleep 1
C.new.test
end
FooExt.class_eval(&f)                    #=> :foo (expected)
BarExt.class_eval(&f)                    #=> :bar (expected)
[ Thread.new { FooExt.class_eval(&f) },  #=> :foo (expected)
Thread.new { BarExt.class_eval(&f) }    #=> :foo (not expected)
].each {|t| t.join }
```

--
Yusuke Endoh mame@tsg.ne.jp

=end

#56 - 12/03/2010 02:05 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/30 Charles Oliver Nutter headius@headius.com:

> - "using" not being a keyword requires all calls to check for refinements all the time, globally degrading performance.

This means that you should check a flag or something in StaticScope
for every method invocation, and you cannot accept the overhead,
right?

What do you think of refine? Should it also be a keyword?
How about refinements activation in reopened definitions and
refinement inheritance?
Can all they be problems?

- instance_eval propagating refinements requires all block invocations to localize what refinements they use for invocation on every activation.
- shared, mutable structures can't be used to store the active refinement, due to concurrency issues
- there are very likely many more complexities than illustrated here that result from the combination of runtime-mutable lexical scoping structures, concurrency, and method caching.

As Yusuke showed in [ruby-core:33535], the current implementation has
this problem.
The current implementation checks only the (singleton) class of the
receiver and the VM version. It should also check the refinements in
cref to avoid this problem,
but it causes more overhead.

I might withdraw the proposal of refinement propagation for blocks
given to instance_eval,
but what do you think of instance_eval without blocks?

x.instance_eval("...")

And, how about to introduce a new method (e.g.,
Module#refinement_eval) which copies a given block and make the copy
refinement aware?
I think blocks are useful to implement DSLs like RSpec.

it "should ..." do
foo = Foo.new
foo.should == ... # Object#should is available only in the block.
end

--
Shugo Maed

=end

#57 - 12/03/2010 02:35 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/11/30 Jedediah Smith redmine@ruby-lang.org:

> There are many ways for refinements to leak out of the lexical scope that uses them:

That is why I called it *pseudo*-lexical.

> 1. Inheritance (which is used in many APIs e.g. ActiveRecord::Base)

I guess this needs more discussion.
We may need something like `private using' which does not affect subclasses.

> 1. DSLs that use instance_eval
> 2. Re-opening a class

I guess they are more controllable than inheritance. But they may
have other problems as pointed out by Charles.

> 1. When a class that uses refinements is itself refined (or does it not work that way?)

I guess it doesn't work that way because we have no local rebinding.

> Also, if "using" and "include" both operate on modules, the difference between them is going to be confusing, particularly for those unfamiliar with the history of the language. It seems arbitrary to overload modules with a use unrelated to their existing purpose. And having to group refinements in modules seems overly complicated for many cases. I would like to be able to do something like this:

I don't think it's so confusing. It complies with the Large Class Principle.

--
Shugo Maeda

=end

#58 - 12/03/2010 02:55 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/2 Yusuke ENDOH mame@tsg.ne.jp:

> > My proposal is to use modules as namespaces for refinements. So
> > indentation is a necessary evil. Otherwise, we need syntax like Java
> > packages and one file for each package.
>
> I'm not against using modules as namespaces and refinement scope,
> but I don't like to see the same module being used for refinement
> and traditional use at the same time.
> Modules for mix-in, modules for collection of helper methods, and
> modules for refinement should be separated, at least, in casual
> use.

I agree with you. But I don't think Ruby should enforce it.

> We can clearly see that the following module FooExt is only for
> refinement. Thus I like this style.
>
> ```
>  module FooExt
>    refine Foo
>    def ... end
>  end
> ```

I don't like this style as the primary way because it can be used for
only one class.
A namespace should be able to have multiple names.

> > > A longer (more "meta-programming-like") name would be appropriate to
> > > the block style, such as Module#refine_class, #refine_class_eval,
> > > #class_eval_with_refinement.
> >
> > Meta-programming means programming on programs, so the non-block style
> > is also a meta-programming feature. Why should only the block style
> > be named more "meta-programming-like"?
>
> I can call it "not for casual use" instead of "meta-programming-
> like."

I see.

> > Could you tell me why
> >
> > ```
> >  refine Foo
> >    def bar; end
> >  end
> > ```
> >
> > is good but
> >
> > ```
> >  refine Foo do
> >    def bar; end
> >  end
> > ```
> >
> > is not so good?
>
> Because block includes method definition. A block looks to me
> "dynamic" behavior, while method definition (using `def' keyword)
> looks "static" behavior.
> Of course I know that both are also evaluated dynamically in
> Ruby, but I don't think that Ruby encourages such a style so much.

method definitions don't look static for me, but nari said the same....
My brain might be too optimized for Ruby.

>> I don't know why meta-programming features should have long names.
>> In Ruby, meta-programming is encouraged, and meta-programming features
>> sometimes have a short name such as eval, but rarely have a keyword.

> Ah, we found the perception gap between I and you.
> I do NOT think that meta-programming is so encouraged even in Ruby.
> It is like "a trick," and should be used only when it is appropriate.
> If it was really so encouraged, Ruby would not provide many
> syntax, such as def' andclass.'
>
> And I guess that "eval" just came from Perl.

I guess it's a problem of the term meta-programming, so let's not use
the word. What I want to say is that I think use frequency of refine
is not so high to require a new keyword, but enough high to have a
nice method name.

To tell the truth, I can accept new keywords refine and using, but I'm
afraid it makes Refinements a Ruby 2.0 feature.

--
Shugo Maeda

=end

#59 - 12/03/2010 03:29 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/2 Haase, Konstantin Konstantin.Haase@student.hpi.uni-potsdam.de:

> I think in general there are two rather distinct use case groups: Those where I don't know and don't want to have to care about the internals of the class that's being refined and those where I do and explicitly want to reach in deep to change a single internal (say in Rails I want to change how class names are mapped to files only in one initializer). If I don't have local rebinding, I would still have to care about the original implementation in order to figure out what methods are calling the method I want to change. In the Enumerable example I would have to override about every method, not only each, in order to change the behavior consistently.

I admit that local rebinding is useful in some cases.

However, Ruby already has dynamic way to extend classes such as monkey
patching and singleton methods, so I guess refinements should be more
(pseudo) static.

--
Shugo Maeda

=end

#60 - 12/03/2010 03:46 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/2 Loren Segal lsegal@soen.ca:

>> Could you tell me why
>>
>> ```
>>  refine Foo
>>    def bar; end
>>  end
>> ```
>>
>> is good but
>>
>> ```
>>  refine Foo do
>>    def bar; end
>>  end
>> ```
>>
>> is not so good?

> The "refine do ... end" block implies a method call which therefore becomes
> easily overridable at run time. This means the compiler cannot statically
> compute refinements unless it just assumed "refine" was never overridden. If
> refinements are truly meant to be lexically scoped, this should be reflected
> in the compiler's handling of them. Charles Nutter's post illustrates why
> this might matter.

Even if using is a keyword, the compiler cannot compute refinements because
refinements are defined at run-time.
The compiler can only know whether refinements are active or not.
However, it may be important for localizing the performance hit to
only scopes where refinements are active, as Charles says.

If the overhead of this check is enough small to apply for every
method invocation, there is no need to make using a keyword.

--
Shugo Maeda

=end

#61 - 12/03/2010 07:20 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/12/3 Shugo Maeda shugo@ruby-lang.org:

> 2010/12/2 Yusuke ENDOH mame@tsg.ne.jp:
>
> > > My proposal is to use modules as namespaces for refinements. So
> > > indentation is a necessary evil. Otherwise, we need syntax like Java
> > > packages and one file for each package.
>
> > I'm not against using modules as namespaces and refinement scope,
> > but I don't like to see the same module being used for refinement
> > and traditional use at the same time.
> > Modules for mix-in, modules for collection of helper methods, and
> > modules for refinement should be separated, at least, in casual
> > use.
>
> I agree with you. But I don't think Ruby should enforce it.

Yes. In fact, I'm ok even if the block style API is *also* included.
However, Ruby can (and should) "navigate" user to encouraged style,
by making encouraged API more useful, such as using shorter method
name.

> > We can clearly see that the following module FooExt is only for
> > refinement. Thus I like this style.
> >
> > module FooExt
> > refine Foo
> > def ... end
> > end
>
> I don't like this style as the primary way because it can be used for
> only one class.

We can use the block style API for such a use case:

```
module FooBarBazExt
  refine_eval Foo do
    def ext; end
  end
  refine_eval Bar do
    def ext; end
  end
  refine_eval Baz do
    def ext; end
  end
end
```

However, it would be better to separate modules, especially when the
code grows:

```
module FooExt
  refine Foo
  def ext1; end
  def ext2; end
  def ext3; end
  ...
end
module BarExt
  refine Bar
  ...
end
module BaZExt
  refine Baz
  ...
end

module FooBarBazExt
  using FooExt # or include FooExt
  using BarExt
  using BazExt
end
```

... Oops! This does not work as excepted. I had believed that this
would work... Why don't you allow this?

> > Because block includes method definition. A block looks to me
> > "dynamic" behavior, while method definition (using `def' keyword)
> > looks "static" behavior.
> > Of course I know that both are also evaluated dynamically in
> > Ruby, but I don't think that Ruby encourages such a style so much.
>
> method definitions don't look static for me, but nari said the same....
> My brain might be too optimized for Ruby.

Many men, many (pseudo) Ruby. My (and nari's) Ruby could be wrong.
Only matz has the true Ruby. We should ask his opinion.

> To tell the truth, I can accept new keywords refine and using, but I'm
> afraid it makes Refinements a Ruby 2.0 feature.

Your suggestion is really well-conceived, but still needs discussion.
If it is included in 1.9.x once, we cannot change the interface because
of compatibility. Thus, we should take cautious steps to include the
feature.

As a first step, how about including only the mechanism and its
undocumented C API, and publishing a refinement gem for Ruby API?
Then, trunk developers can easily examine the feature by using the gem.
We can also examine the actual performance. If we find any problem or
came up with a better idea, we can change the API without care of
compatibility (because it is just an undocumented API and an untrustful
gem!), or even remove and forget it at worst.

Of course, this is a topic after the discussion is closed.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#62 - 12/03/2010 10:05 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/3 Yusuke ENDOH mame@tsg.ne.jp:

> > > I'm not against using modules as namespaces and refinement scope,
> > > but I don't like to see the same module being used for refinement
> > > and traditional use at the same time.
> > > Modules for mix-in, modules for collection of helper methods, and
> > > modules for refinement should be separated, at least, in casual
> > > use.
>
> > I agree with you. But I don't think Ruby should enforce it.
>
> Yes. In fact, I'm ok even if the block style API is *also* included.
> However, Ruby can (and should) "navigate" user to encouraged style,
> by making encouraged API more useful, such as using shorter method
> name.

Hmm, I prefer the keyword refine to the method refine without blocks....

> > I don't like this style as the primary way because it can be used for
> > only one class.
>
> We can use the block style API for such a use case:
>
> ```
>  module FooBarBazExt
>    refine_eval Foo do
>      def ext; end
>    end
>    refine_eval Bar do
>      def ext; end
>    end
>    refine_eval Baz do
>      def ext; end
>    end
>  end
> ```

refine_eval doesn't look a good name. I don't think the above code
should be so discouraged.

> However, it would be better to separate modules, especially when the
> code grows:
>
> ```
>  module FooExt
>    refine Foo
>    def ext1; end
>    def ext2; end
>    def ext3; end
>    ...
>  end
>  module BarExt
>    refine Bar
>    ...
>  end
>  module BaZExt
>    refine Baz
>    ...
>  end
>
>  module FooBarBazExt
>    using FooExt # or include FooExt
>    using BarExt
>    using BazExt
>  end
> ```
>
> ... Oops! This does not work as excepted. I had believed that this
> would work... Why don't you allow this?

Currently refine doesn't work without blocks, but do you mean that?

> > To tell the truth, I can accept new keywords refine and using, but I'm

> > afraid it makes Refinements a Ruby 2.0 feature.
>
> Your suggestion is really well-conceived, but still needs discussion.
> If it is included in 1.9.x once, we cannot change the interface because
> of compatibility. Thus, we should take cautious steps to include the
> feature.
>
> As a first step, how about including only the mechanism and its
> *undocumented* C API, and publishing a refinement gem for Ruby API?
> Then, trunk developers can easily examine the feature by using the gem.
> We can also examine the actual performance. If we find any problem or
> came up with a better idea, we can change the API without care of
> compatibility (because it is just an undocumented API and an untrustful
> gem!), or even remove and forget it at worst.

I think it's hard to add new keywords by a gem. Have you abandoned
keywords? I prefer to the keyword refine to the method refine without
blocks suggested by you.

Isn't it enough to introduce refinements as an experimental feature,
at least in trunk?

--
Shugo Maeda

=end

#63 - 12/03/2010 11:24 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/12/3 Shugo Maeda shugo@ruby-lang.org:

> > ... Oops! This does not work as excepted. I had believed that this
> > would work... Why don't you allow this?
>
> Currently refine doesn't work without blocks, but do you mean that?

Ah sorry.

module FooExt
refine(Fixnum) do
def /(other); quo(other); end
end
end
module BarExt
include FooExt # or using FooExt
end
using BarExt
p 1 / 2 #=> actual: 0, expected: (1/2)

> I think it's hard to add new keywords by a gem. Have you abandoned
> keywords? I prefer to the keyword refine to the method refine without
> blocks suggested by you.

I first said in [ruby-core:33375]:

> However, we should discuss this topic (new keyword) towards
> 2.0. Module's methods are not bad, as a part of reflection
> features (such as Module#define_method for `def' keyword).

Needless to say, we must not add any new keywords to 1.9.x, especially
normal simple word like "refine." I'm ok to include this feature in
1.9.x, and now I believe that gem is a good idea as the first step.

However, I received many negative comments to this approach (gem),
from nars*, kosak*, ko*, nakad*, shyouhe*. They seem to think that
it is better to import your patch "as is".

> Isn't it enough to introduce refinements as an experimental feature,
> at least in trunk?

I don't like to include a feature called "experimental", not because
it is not complete yet, but because it becomes "de facto standard."
It would be good if there are not only "document" but also "mechanism"
to inform users that the feature is experimental, such as warning or
a new method like "RubyVM.enable_experimental_features".

Note that this is just my opinion, and that I seem to be in the
minority ;-)
I hope that 1.9.x would be stable, but many other committers seem to
hope to include new feature in 1.9.x aggressively.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#64 - 12/04/2010 09:31 PM - headius (Charles Nutter)

=begin
Hello,

On Thu, Dec 2, 2010 at 11:05 PM, Shugo Maeda shugo@ruby-lang.org wrote:

> 2010/11/30 Charles Oliver Nutter headius@headius.com:
>
> > - "using" not being a keyword requires all calls to check for refinements all the time, globally degrading performance.
>
> This means that you should check a flag or something in StaticScope
> for every method invocation, and you cannot accept the overhead,
> right?

Every little bit matters. In experimental optimizations, JRuby is able
to reduce a dynamic call to two memory indirections + compare + static
dispatch directly to jitted code. When the dispatch path is this fast,
adding multiple additional layers of memory indirection and comparison
to support rarely-changing refinements can definitely show up. I also
have not attempted to implement an optimized version of refinement
dispatch, and I worry that there will be additional
performance-impacting issues.

> What do you think of refine? Should it also be a keyword?
> How about refinements activation in reopened definitions and
> refinement inheritance?
> Can all they be problems?

refine as keyword: I don't think so, since it, like many other
meta-programming methods, applies its changes only once to the
class/module that surrounds it and the block it has been given. It's
very "magical", but I'm not sure that's enough of a reason to make it
a keyword.

refinement activation in reopened definition: Reopened definitions are
hierarchies of new lexical scopes. If a refinement is active for those
scopes it will only affect them and no other scopes. Obviously you
should not be able to apply a new refinement to scopes that have
already been closed without refinements in place. That is my #1
concern with this proposal...what is static (as in StaticScope in
JRuby) should remain static. I don't believe "refine" or refinements
applied in reopened classes violate that rule.

> > - there are very likely many more complexities than illustrated here that result from the combination of runtime-mutable lexical scoping structures, concurrency, and method caching.
>
> As Yusuke showed in [ruby-core:33535], the current implementation has
> this problem.
> The current implementation checks only the (singleton) class of the
> receiver and the VM version. It should also check the refinements in

> cref to avoid this problem,
> but it causes more overhead.

More overhead is always bad if it affects performance globally :)

> I might withdraw the proposal of refinement propagation for blocks
> given to instance_eval,
> but what do you think of instance_eval without blocks?
>
> ```
>  x.instance_eval("...")
> ```

This form is "safe" since the string needs to be parsed and evaluated
(and provided with a new scope) each time. I worry a bit about the
inconsistency of having the "" form propagate refinements but the {}
form not propagating refinements. Perhaps this is a big like the
duality of constant lookup inside instance_eval?

> And, how about to introduce a new method (e.g.,
> Module#refinement_eval) which copies a given block and make the copy
> refinement aware?
> I think blocks are useful to implement DSLs like RSpec.
>
> ```
>  it "should ..." do
>    foo = Foo.new
>    foo.should == ...  # Object#should is available only in the block.
>  end
> ```

Yes, I agree this is a useful and difficult case to address. One
possible solution would be making the modification of the block
explicit and permanent:

```
p = proc { bar }
p.refine! BarModifyingExt
# p from now on has BarModifyingExt's refinements applied to it
```

This still can suffer from concurrency problems, if the "p" proc is
stored somewhere and in use when Proc#refine is called, but I think it
addresses the problem of having blocks flip back and forth between
refined and unrefined. It's not perfect wrt concurrency, but it's
(somewhat) better.

I'm not sure this addresses the problems with performance, though. All
method calls in all blocks everywhere would still need to check if
they've been turned into a Proc and "refined", and that would be a
larger impact than simply checking a global flag.

The bottom line is that anything which makes static parse output now
mutable will always be a concurrency problem, and probably always be a
performance hit. All Ruby implementations currently depend on certain
immutable truths about parser output, and this proposal violates many
of those truths.

To be honest, I think it should be a using directive at the file's
toplevel, so subsequently-parsed blocks know they're going to be
refined. I believe more and more that "using" needs to be a keyword;
having yet another "magic" method that can alter parse/execution
behavior beyond its scope scares me.

```
require 'rspec'

# RSpec::Refinements would define all methods currently
# defined against Object or Kernel, and the refine here
# would simply apply them to this file.
# TODO: using affects current scope or only child scopes?
# The latter would be best, since the current scope is already
# "active" and should not be modified.
using RSpec::Refinements
# does this scope have refinements active?

# describe is from refinement(?) or a real method
describe 'blah' do
# this scope has refinements active
```

"it" is either from refinement or a real method
it 'blah' do
# this scope has refinements active

```
# "should" is from refinement
x.should == y
```

end
end

- Charlie

=end

#65 - 12/04/2010 09:32 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/3 Yusuke ENDOH mame@tsg.ne.jp:

> > > ... Oops! This does not work as excepted. I had believed that this
> > > would work... Why don't you allow this?
>
> > Currently refine doesn't work without blocks, but do you mean that?
>
> Ah sorry.
>
> ```
> module FooExt
>   refine(Fixnum) do
>     def /(other); quo(other); end
>   end
> end
> module BarExt
>   include FooExt # or using FooExt
> end
> using BarExt
> p 1 / 2 #=> actual: 0, expected: (1/2)
> ```

I forgot to support it. Please apply the attached patch. I have also
added Kernel#refine.

> > I think it's hard to add new keywords by a gem. Have you abandoned
> > keywords? I prefer to the keyword refine to the method refine without
> > blocks suggested by you.
>
> I first said in [ruby-core:33375]:
>
> > However, we should discuss this topic (new keyword) towards
> > 2.0. Module's methods are not bad, as a part of reflection
> > features (such as Module#define_method for `def' keyword).
>
> Needless to say, we must not add any new keywords to 1.9.x, especially
> normal simple word like "refine." I'm ok to include this feature in
> 1.9.x, and now I believe that gem is a good idea as the first step.
>
> However, I received many negative comments to this approach (gem),
> from nars*, kosak*, ko*, nakad*, shyouhe*. They seem to think that
> it is better to import your patch "as is".

I also wouldn't like to see such a language core feature in a gem.

> > Isn't it enough to introduce refinements as an experimental feature,
> > at least in trunk?
>
> I don't like to include a feature called "experimental", not because
> it is not complete yet, but because it becomes "de facto standard."
> It would be good if there are not only "document" but also "mechanism"

> to inform users that the feature is experimental, such as warning or
> a new method like "RubyVM.enable_experimental_features".

As far as I know, the current status of trunk is unstable, and if new
features break backward compatibility, Yugui will create the branch
ruby_1_9, won't you, Yugui? So, we don't need the above mechanism
until new experimental features are decided to include to 1.9.x.

> Note that this is just my opinion, and that I seem to be in the
> minority ;-)
> I hope that 1.9.x would be stable, but many other committers seem to
> hope to include new feature in 1.9.x aggressively.

I also hope that 1.9.x would be stable, but I'd like to develop
aggressively in trunk. I don't think the current design and
implementation of refinements are mature, but I can't make them mature
on my own, so I'd like to have help from other committers.

--
Shugo Maeda

Attachment: indirect_using_and_kernel_refine_20101204.diff
=end

#66 - 12/04/2010 09:48 PM - headius (Charles Nutter)

=begin
Hello,

On Thu, Dec 2, 2010 at 9:42 PM, Yusuke ENDOH mame@tsg.ne.jp wrote:

> 2010/11/30 Charles Oliver Nutter headius@headius.com:
>
>> The global serial number approach in 1.9 means that any change that
>> flip that serial number cause all caches everywhere to invalidate.
>> Normally this only happens on method definition or module inclusion,
>> which is why defining methods or including modules at runtime is
>> strongly discouraged for performance reasons.
>
> Sorry I'm not sure that I could follow your argument about performance,
> so I may miss your point.
>
> I guess that casual users will execute all refinements immediately after
> program is started, like class definition and method definition.
> Thus, the global serial number approach will work well for refinement
> in main use cases, I think.
> In the sense, nested function by using refinements may be a problem.

For the main cases, I see it working this way:

- Parser sees "using" in a scope
- Child scopes will now be parsed as though they are refined
- VCALL, FCALL, CALL in child scopes are instead RVCALL, RFCALL, RCALL ** Likely other calls must be replaced/wrapped too, e.g. +=, []=, []+=, and so on. (messy!)
- Refined call versions check refinement first before checking metaclass for target method

Maybe this helps make it clear why the refinement state of a given
scope must never be mutable...we want to be able to limit the extra
refinement checks to calls where refinements are active, leaving
unrefined calls as they are today.

>> And one last case that's a problem: author's intent. If I write a
>> block of code that does "1 + 1", I intend for that code to do normal
>> Fixnum#+, and I intend for the result to be 2. It should not be
>> possible for a caller to change the intent of my code just because I
>> passed it in a bock. This has been my argument against using blocks as
>> bindings, and it's another argument against instance_eval being able
>> to force refinments into "someone else's code".
>
> This is not a problem, but rather improvement. There is already open

> class which so often breaks your intent. Refinements may also break
> your intent, but it is less often and more controllable than open class.

Open classes break my intent globally and usually at boot time.
Refinement propagation into blocks is much more likely to break my
intent at some arbitrary time in the future, since refinements are
much more lazily applied than open class modifications. Lazy breakage
is always worse than eager breakage.

>> Now, some positive reinforcement for "using" being a keyword and
>> instance_eval not propagating refinements.

> I'm not against your proposal, but I wonder if it does not make sense
> because we can still write: eval("using FooExt")

eval is not a concern, because it always creates a new scope. If a
"using" is active, those scopes (or at least their child scopes) would
be statically marked as "refined", and my requirements are still met.

> To address your concern, `using' keyword should have a block:
>
> ```
>  using FooExt
>    # FooExt enabled
>  end
>  # FooExt disabled
> ```
>
> I don't like this syntax because of more indentation, though.

I mention this at the bottom of my reply to Shugo...I'm uncomfortable
with "using" applying to the current scope and not just to child
scopes encountered after it. Once code begins executing against a
given scope, that scope's static state (like whether a refinement is
active) should never change. Given that "using" occurs only at
toplevel or in class/module bodies (i.e., during file load/require),
there's not as many concurrency concerns here. There are just a lot of
messy issues with "using" applying to the current scope:

- Does it affect calls that came before it?
- Do multiple "using" directives combine or overwrite each other?
- When pre-parsing or pre-compiling (AOT stuff), you would have to walk the whole file looking for "using" to know how to emit compiled results.

What we're really talking about with refinements is a way to lexically
say "make calls in these scopes do an extra indirection during
lookup." Unrefined calls should never have to perform this additional
indirection.

>> I can try to come up with a concrete example of the problems with the
>> current proposal and implementation, but the concurrency cases would
>> be difficult to show.

> I might find serious concurrency problem of Shugo's patch, though I'm
> not sure that this is what you mean. Indeed, we may have to give up
> propagating refinements via block.
>
> ```
>  class C
>    def test; p :test; end
>  end
>  module FooExt
>    refine(C) { def test; p :foo; end }
>  end
>  module BarExt
>    refine(C) { def test; p :bar; end }
>  end
>  f = proc do
>    sleep 1
>    C.new.test
>  end
>  FooExt.class_eval(&f)                  #=> :foo (expected)
>  BarExt.class_eval(&f)                  #=> :bar (expected)
>  [ Thread.new { FooExt.class_eval(&f) },  #=> :foo (expected)
> ```

```
  Thread.new { BarExt.class_eval(&f) }   #=> :foo (not expected)
].each {|t| t.join }
```

Thank you, this helps illustrate the problems with modifying a block's (formerly) static state. We really must not modify an already-parsed/compiled scope with new refinements.

- Charlie

=end

#67 - 12/04/2010 10:52 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/3 Yusuke ENDOH mame@tsg.ne.jp:

> I might find serious concurrency problem of Shugo's patch, though I'm not sure that this is what you mean. Indeed, we may have to give up propagating refinements via block.

I have fixed it with additional overhead:)
Please try the attached patch.

--
Shugo Maeda

Attachment: inline_cache_fix_20101204.diff
=end

#68 - 12/04/2010 11:19 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/4 Charles Oliver Nutter headius@headius.com:

> > > - "using" not being a keyword requires all calls to check for refinements all the time, globally degrading performance.
> >
> > This means that you should check a flag or something in StaticScope for every method invocation, and you cannot accept the overhead, right?
>
> Every little bit matters. In experimental optimizations, JRuby is able to reduce a dynamic call to two memory indirections + compare + static dispatch directly to jitted code. When the dispatch path is this fast, adding multiple additional layers of memory indirection and comparison to support rarely-changing refinements can definitely show up. I also have not attempted to implement an optimized version of refinement dispatch, and I worry that there will be additional performance-impacting issues.

I see.

> > What do you think of refine? Should it also be a keyword?
> > How about refinements activation in reopened definitions and refinement inheritance?
> > Can all they be problems?
>
> refine as keyword: I don't think so, since it, like many other meta-programming methods, applies its changes only once to the class/module that surrounds it and the block it has been given. It's very "magical", but I'm not sure that's enough of a reason to make it a keyword.

I'm also not sure.

refinement activation in reopened definition: Reopened definitions are
hierarchies of new lexical scopes. If a refinement is active for those
scopes it will only affect them and no other scopes. Obviously you
should not be able to apply a new refinement to scopes that have
already been closed without refinements in place. That is my #1
concern with this proposal...what is static (as in StaticScope in
JRuby) should remain static. I don't believe "refine" or refinements
applied in reopened classes violate that rule.

I see.

- there are very likely many more complexities than illustrated here that result from the combination of runtime-mutable lexical scoping structures, concurrency, and method caching.

As Yusuke showed in [ruby-core:33535], the current implementation has
this problem.
The current implementation checks only the (singleton) class of the
receiver and the VM version. It should also check the refinements in
cref to avoid this problem,
but it causes more overhead.

More overhead is always bad if it affects performance globally :)

I thought you might say that, but I did it ;)

I might withdraw the proposal of refinement propagation for blocks
given to instance_eval,
but what do you think of instance_eval without blocks?

```
 x.instance_eval("...")
```

This form is "safe" since the string needs to be parsed and evaluated
(and provided with a new scope) each time. I worry a bit about the
inconsistency of having the "" form propagate refinements but the {}
form not propagating refinements. Perhaps this is a big like the
duality of constant lookup inside instance_eval?

I think the inconsistency is not a problem because the "" form and the
{} form are already inconsistent as to constant lookup as you say.

And, how about to introduce a new method (e.g.,
Module#refinement_eval) which copies a given block and make the copy
refinement aware?
I think blocks are useful to implement DSLs like RSpec.

```
 it "should ..." do
   foo = Foo.new
   foo.should == ...  # Object#should is available only in the block.
 end
```

Yes, I agree this is a useful and difficult case to address. One
possible solution would be making the modification of the block
explicit and permanent:

p = proc { bar }
p.refine! BarModifyingExt

p from now on has BarModifyingExt's refinements applied to it

It's an interesting idea.

To be honest, I think it should be a using directive at the file's
toplevel, so subsequently-parsed blocks know they're going to be
refined. I believe more and more that "using" needs to be a keyword;
having yet another "magic" method that can alter parse/execution
behavior beyond its scope scares me.

I understand your worry. But refinements are still immature in other
aspects than performance, so I'd like to experiment on refinements
before improving performance. Methods are easier than keywords to do
it.

--
Shugo Maeda

=end

#69 - 12/05/2010 02:56 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33568] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Sat, 4 Dec 2010 21:48:00 +0900, Charles Oliver Nutter headius@headius.com writes:

|For the main cases, I see it working this way:
|
|* Parser sees "using" in a scope
|* Child scopes will now be parsed as though they are refined
|* VCALL, FCALL, CALL in child scopes are instead RVCALL, RFCALL, RCALL

I see this optimization nice for refinement, but it's impossible to
implement local rebinding (a la classbox) that some wanted in this
thread this way.

```
                            matz.
```

=end

#70 - 12/05/2010 10:07 AM - headius (Charles Nutter)

=begin
Hello,

On Sat, Dec 4, 2010 at 11:56 AM, Yukihiro Matsumoto matz@ruby-lang.org wrote:

> |For the main cases, I see it working this way:
> |
> |* Parser sees "using" in a scope
> |* Child scopes will now be parsed as though they are refined
> |* VCALL, FCALL, CALL in child scopes are instead RVCALL, RFCALL, RCALL
>
> I see this optimization nice for refinement, but it's impossible to
> implement local rebinding (a la classbox) that some wanted in this
> thread this way.

Yes, and that is a primary reason why I would not include local
rebinding if I were implementing refinements. I don't see how it's
possible to do without adding overhead to every call (to constantly
check if there's a refinement active) and forcing the global cache to
be flushed repeatedly (since a refinement can be applied from any
scope in any thread at any time). In essence, with local rebinding,
all method calls would have to:

- Ping on every call to see if a refinement is active, since it's not possible to narrow the refinement to only locally-rebound methods.
- If a refinement is active, check to see if any active refinements affect them. This would at minimum require inspecting the refinement to see if it includes methods for the target object's class.
- Either only cache refined methods within that one activation or don't cache methods at all, since other calls shouldn't see refined methods in the global cache.
- Do all this in a way that is thread-safe, does not modify any globally-visible state, and only impacts the activation where refinements are active.

So at a minimum, all calls in the system would have to constantly ping
additional some global state. Because any refinement anywhere would
modify that state, any time any thread does a local rebinding, all
calls would have to check to see if they are affected. And refined
code would require at least adding extra checks to all method cache
validations (to ensure what was cached was not from a refinement not
active in the current activation) or require that refined code never
cache (or only cache within that activation, which would only be
useful if those call sites are encountered many times in a single

activation).

For a real-world example of how local rebinding causes problems, look
at Groovy. Groovy allows local rebinding down-thread; i.e. if a
"category" is active, all invocations deeper in that thread must check
the thread-local category to see if they are affected by it. As a
result:

- All calls everywhere have to constantly check thread-local state to see if a category is active
- Calls down-stream from a category have to check whether the category affects them
- Calls down-stream from a category that are affected by it have to do a slow-path uncached method lookup every time

The end result of Categories in Groovy are:

- All performance is affected by them, and fixing it is difficult
- Users don't use them anymore, since the performance of calls down-stream from the category are much slower than regular calls
- The Groovy team generally regrets that they were added, due to the long-term effects

They are considered (by most folks I've talked to) to be a mistake,
but now there's no going back. It's interesting to note that the
Groovy approach at least only affects one thread, and there are no
concurrency issues; only the categorized thread sees the change, and
since it's explicitly in thread-local state, there's no impact across
threads. But as in Refinements, the presence of Categories as a
feature means all calls have to be slower. And even worse, as Groovy
has moved forward with more and more optimizations, two things have
made it difficult: open classes and categories. The latter could have
been avoided.

A Refinements implementation that adds overhead to all calls will
damage Ruby (or at least, Ruby's chances of being a high-performance
language) forever.

- Charlie

=end

#71 - 12/05/2010 10:38 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

I am neutral about local rebinding. It is useful sometimes (as
someone has pointed out in this thread). At the same time, I agree
with you that performance impact is a huge negative factor against
local rebinding. We need more input. Dave, what do you think?
Anyone else?

```
                                matz.
```

In message "Re: [ruby-core:33578] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Sun, 5 Dec 2010 10:06:46 +0900, Charles Oliver Nutter headius@headius.com writes:
|
|Hello,
|
|On Sat, Dec 4, 2010 at 11:56 AM, Yukihiro Matsumoto matz@ruby-lang.org wrote:
|> |For the main cases, I see it working this way:
|> |
|> |* Parser sees "using" in a scope
|> |* Child scopes will now be parsed as though they are refined
|> |* VCALL, FCALL, CALL in child scopes are instead RVCALL, RFCALL, RCALL
|>
|> I see this optimization nice for refinement, but it's impossible to
|> implement local rebinding (a la classbox) that some wanted in this
|> thread this way.
|
|Yes, and that is a primary reason why I would not include local
|rebinding if I were implementing refinements. I don't see how it's
|possible to do without adding overhead to every call (to constantly
|check if there's a refinement active) and forcing the global cache to
|be flushed repeatedly (since a refinement can be applied from any
|scope in any thread at any time). In essence, with local rebinding,
|all method calls would have to:
|
|* Ping on every call to see if a refinement is active, since it's not

|possible to narrow the refinement to only locally-rebound methods.
|* If a refinement is active, check to see if any active refinements
|affect them. This would at minimum require inspecting the refinement
|to see if it includes methods for the target object's class.
|* Either only cache refined methods within that one activation or
|don't cache methods at all, since other calls shouldn't see refined
|methods in the global cache.
|* Do all this in a way that is thread-safe, does not modify any
|globally-visible state, and only impacts the activation where
|refinements are active.
|
|So at a minimum, all calls in the system would have to constantly ping
|*additional* some global state. Because any refinement anywhere would
|modify that state, any time any thread does a local rebinding, all
|calls would have to check to see if they are affected. And refined
|code would require at least adding extra checks to all method cache
|validations (to ensure what was cached was not from a refinement not
|active in the current activation) or require that refined code never
|cache (or only cache within that activation, which would only be
|useful if those call sites are encountered many times in a single
|activation).
|
|For a real-world example of how local rebinding causes problems, look
|at Groovy. Groovy allows local rebinding down-thread; i.e. if a
|"category" is active, all invocations deeper in that thread must check
|the thread-local category to see if they are affected by it. As a
|result:
|
|* All calls everywhere have to constantly check thread-local state to
|see if a category is active
|* Calls down-stream from a category have to check whether the category
|affects them
|* Calls down-stream from a category that are affected by it have to do
|a slow-path uncached method lookup every time
|
|The end result of Categories in Groovy are:
|
|* All performance is affected by them, and fixing it is difficult
|* Users don't use them anymore, since the performance of calls
|down-stream from the category are much slower than regular calls
|* The Groovy team generally regrets that they were added, due to the
|long-term effects
|
|They are considered (by most folks I've talked to) to be a mistake,
|but now there's no going back. It's interesting to note that the
|Groovy approach at least only affects one thread, and there are no
|concurrency issues; only the categorized thread sees the change, and
|since it's explicitly in thread-local state, there's no impact across
|threads. But as in Refinements, the presence of Categories as a
|feature means all calls have to be slower. And even worse, as Groovy
|has moved forward with more and more optimizations, two things have
|made it difficult: open classes and categories. The latter could have
|been avoided.
|
|A Refinements implementation that adds overhead to all calls will
|damage Ruby (or at least, Ruby's chances of being a high-performance
|language) forever.
|
|- Charlie

=end

#72 - 12/06/2010 12:16 PM - raggi (James Tucker)

=begin

On Dec 4, 2010, at 5:37 PM, Yukihiro Matsumoto wrote:

> Hi,
>
> I am neutral about local rebinding. It is useful sometimes (as someone has pointed out in this thread). At the same time, I agree with you that performance impact is a huge negative factor against local rebinding. We need more input. Dave, what do you think? Anyone else?

```
                            matz.
```

I don't think it's just performance, it's debugging too.

If something fails due to a data error introduced mid-stack due to an unexpected refinement then you have to walk up the stack checking to see what refinements are used. This means I can't use irb with a loaded environment and reflection anymore. I have to use a debugger and walk up frames reflecting on the current state of the world.

At best you can add the refinement to the backtrace (I have no idea if this is done already? - sorry haven't been following in detail). This will only help if the errored refinement method is actually in the stack, if it isn't you have to walk around odd places in the program / libraries being sure that you're clear what refinements are where. I'm struggling to see how debugging this is any easier than debugging monkey patching?

=end

#73 - 12/06/2010 02:09 PM - wycats (Yehuda Katz)

=begin
I disagree with this argument. If the feature behaved as you expected, you
would never be able to include a refinement into a scope without reading the
implementation of all callees to make sure that the refinements you included
do not conflict with methods used downstream.

Consider this (very simplified) scenario:

```
module ActiveSupport
refine String do
def camelize
dup.split(/_/).map{ |word| word.capitalize }.join('')
end
end
end
```

And now consider some class that uses this refinement:

```
class Constantizer
using ActiveSupport

def initialize(str)
@class_name = str.camelize
end

def const
Object.const_get(@class_name)
end
end
```

Now let's say that I have created my own refinement which refines camelize:

```
module MyApp
refine String do
def camelize
"" + capitalize + ""
end
end
end
```

And now I go ahead and use that refinement:

```
class MyApp::Application
using MyApp

def initialize
@cache = {}
end

def cache(name)
@cache[name.camelize] = Constantizer.new(name).const
end

def lookup(name)
@cache[name.camelize]
end
end
```

This illustrates that when using a refinement, you would be forced to
understand the implementation of all callees, to make sure that you are not
accidentally leaking a refinement into a scope that doesn't expect it. Since
the point of encapsulating this functionality is to avoid callers needing to
know the implementation of the callee (and to allow the callee to change its
implementation as long as it doesn't modify the interface), this behavior is
unexpected.

Instead of making it easy to isolate the effects of a refinement to a scope,
those effects could leak into other scopes that do not expect to be mutated
in that way.

Essentially, by leaking the refinement, we are now making every single
method call in the callee a part of the public interface (as you believe
that the fact that #map calls #each is part of the public interface). While
we do have a few informal interfaces that behave this way, the vast majority
of Ruby code does not, and this behavior would render the feature much less
useful, and result in failure to meet the goals of the feature (the ability
to apply a refinement to an area of code without fearing breakage of other
unrelated code).

Yehuda Katz
Architect | Strobe
(ph) 718.877.1325

On Wed, Nov 24, 2010 at 11:45 AM, Dave Thomas redmine@ruby-lang.org wrote:

> Issue #4085 has been updated by Dave Thomas.
>
> module Quux
> using FooExt
>
> foo = Foo.new
> foo.bar # => FooExt#bar
> foo.baz # => Foo#bar
> end
>
> This behavior makes me nervous—I can see arguments for it, but at the same
> time I can see in leading to problems, particularly in well structured
> classes where a large set of behaviors is defined in terms of one method.
> I'm sure the syntax below is wrong, but look at the spirit of it.
>
> module DoubleEach
> refine Array do
> def each
> super do |val|
> yield 2*val
> end
> end
> end
> end
>
> using DoubleEach
>
> [1, 2, 3].each {|v| p v }   #=> 2, 4, 6
>
> [1, 2, 3].min   #=> 1
>
> That would be surprising to me, as I'd expect the behavior of all the
> Enumerable methods, which depend on each, to change if I change the behavior
>
> ## of each.
>
> http://redmine.ruby-lang.org/issues/show/4085
>
> ---
>
> http://redmine.ruby-lang.org

I disagree with this argument. If the feature behaved as you expected, you would never be able to include a refinement into a scope without reading the implementation of all callees to make sure that the refinements you included do not conflict with methods used downstream.
Consider this (very simplified) scenario:module ActiveSupport refine String do   def camelize   dup.split(//).map{ |word| word.capitalize }.join('')   end endendAnd now consider some class that uses this refinement:class Constantizer using ActiveSupport def initialize(str)   @class_name = str.camelize end def const   Object.const_get(@class_name) endendNow let's say that I have created my own refinement which refines camelize:

module MyApp refine String do   def camelize   "" + capitalize + "_"   end endend
And now I go ahead and use that refinement:class MyApp::Application using MyApp def initialize   @cache = {} end
  def cache(name)   @cache[name.camelize] = Constantizer.new(name).const end def lookup(name)   @cache[name.camelize] endend
This illustrates that when using a refinement, you would be forced to understand the implementation of all callees, to make sure that you are not accidentally leaking a refinement into a scope that doesn't expect it. Since the point of encapsulating this functionality is to avoid callers needing to know the implementation of the callee (and to allow the callee to change its implementation as long as it doesn't modify the interface), this behavior is unexpected.
Instead of making it easy to isolate the effects of a refinement to a scope, those effects could leak into other scopes that do not expect to be mutated in that way.Essentially, by leaking the refinement, we are now making every single method call in the callee a part of the public interface (as you believe that the fact that #map calls #each is part of the public interface). While we do have a few informal interfaces that behave this way, the vast majority of Ruby code does not, and this behavior would render the feature much less useful, and result in failure to meet the goals of the feature (the ability to apply a refinement to an area of code without fearing breakage of other unrelated code).
Yehuda KatzArchitect | Strobe(ph) 718.877.1325
On Wed, Nov 24, 2010 at 11:45 AM, Dave Thomas <redmine@ruby-lang.org> wrote:
Issue #4085 has been updated by Dave Thomas.

```
 module Quux
   using FooExt

   foo = Foo.new
   foo.bar  # => FooExt#bar
   foo.baz  # => Foo#bar
 end
```

This behavior makes me nervous—I can see arguments for it, but at the same time I can see in leading to problems, particularly in well structured classes where a large set of behaviors is defined in terms of one method. I'm sure the syntax below is wrong, but look at the spirit of it.

```
 module DoubleEach
   refine Array do
     def each
        super do |val|
          yield 2*val
        end
     end
   end
 end

 using DoubleEach

 [ 1, 2, 3 ].each {|v| p v }   #=> 2, 4, 6

 [ 1, 2, 3 ].min    #=> 1
```

That would be surprising to me, as I'd expect the behavior of all the Enumerable methods, which depend on each, to change if I change the behavior of each.
----
http://redmine.ruby-lang.org/issues/show/4085

----
http://redmine.ruby-lang.org

=end

#74 - 12/06/2010 02:13 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/5 Yukihiro Matsumoto matz@ruby-lang.org:

> I am neutral about local rebinding. It is useful sometimes (as
> someone has pointed out in this thread). At the same time, I agree
> with you that performance impact is a huge negative factor against
> local rebinding. We need more input. Dave, what do you think?
> Anyone else?

I believe we should not support local rebinding.

I guess local rebinding is not designed for refinements of classes
which have many clients like built-in classes. With local rebinding,
when we refine a built-in class such as String, we have to take care

carefully not to break existing code. This means that, it's
ridiculous to use the following refinement with local rebinding.

```
module MathN
refine Fixnum do
def /(other) quo(other) end
end
end
```

My intention is to bring modularity into monkey patching, not to bring
extensibility. We already have enough extensibility in Ruby.

--
Shugo Maeda

=end

#75 - 12/06/2010 02:42 PM - wycats (Yehuda Katz)

=begin
I think that, for this same reason, using should normally not apply
outside of the immediate lexical scope. I actually believed that this was
the default behavior, and explained why I thought it was a good idea on my
blog (http://yehudakatz.com/2010/11/30/ruby-2-0-refinements-in-practice/).

In general, the most utility from refinements comes from being able to use
them freely, without worrying about accidentally breaking other code. In the
case of subclasses, for instance, I would want to be able to add a
refinement in a Rails 3.0.3 class (like ActionController::Base) for
convenience without worrying about breaking existing Rails apps that don't
expect core classes to suddenly change behavior.

I can understand the utility of offering the inherited style when desired,
but I don't think it should be the default. By default, the principle of
this feature requires that refinements modify zero code outside its lexical
scope.

Perhaps a special form of using that would also affect subclasses (for
example, for the case where Rails wants to expose ActiveSupport into its
subclasses):

```
class ActionController::Base
using ActiveSupport, inherited: true
end
```

Yehuda Katz
Architect | Strobe
(ph) 718.877.1325

On Mon, Dec 6, 2010 at 12:12 AM, Shugo Maeda shugo@ruby-lang.org wrote:

> Hi,
>
> 2010/12/5 Yukihiro Matsumoto matz@ruby-lang.org:
>
>> I am neutral about local rebinding. It is useful sometimes (as
>> someone has pointed out in this thread). At the same time, I agree
>> with you that performance impact is a huge negative factor against
>> local rebinding. We need more input. Dave, what do you think?
>> Anyone else?
>
> I believe we should not support local rebinding.
>
> I guess local rebinding is not designed for refinements of classes
> which have many clients like built-in classes. With local rebinding,
> when we refine a built-in class such as String, we have to take care
> carefully not to break existing code. This means that, it's
> ridiculous to use the following refinement with local rebinding.
>
> ```
> module MathN
> refine Fixnum do
> def /(other) quo(other) end
> end
> end
> ```

> My intention is to bring modularity into monkey patching, not to bring
> extensibility. We already have enough extensibility in Ruby.
>
> --
> Shugo Maeda

I think that, for this same reason, using should normally not apply outside of the immediate lexical scope. I actually believed that this was the default behavior, and explained why I thought it was a good idea on my blog (http://yehudakatz.com/2010/11/30/ruby-2-0-refinements-in-practice/).
In general, the most utility from refinements comes from being able to use them freely, without worrying about accidentally breaking other code. In the case of subclasses, for instance, I would want to be able to add a refinement in a Rails 3.0.3 class (like ActionController::Base) for convenience without worrying about breaking existing Rails apps that don't expect core classes to suddenly change behavior.
I can understand the utility of offering the inherited style when desired, but I don't think it should be the default. By default, the principle of this feature requires that refinements modify zero code outside its lexical scope.
Perhaps a special form of using that would also affect subclasses (for example, for the case where Rails wants to expose ActiveSupport into its subclasses):class ActionController::Base
  using ActiveSupport, inherited: trueendYehuda KatzArchitect | Strobe(ph) 718.877.1325
On Mon, Dec 6, 2010 at 12:12 AM, Shugo Maeda <shugo@ruby-lang.org> wrote:
Hi,

2010/12/5 Yukihiro Matsumoto <matz@ruby-lang.org>:
> I am neutral about local rebinding. It is useful sometimes (as
> someone has pointed out in this thread). At the same time, I agree
> with you that performance impact is a huge negative factor against
> local rebinding. We need more input. Dave, what do you think?
> Anyone else?

I believe we should not support local rebinding.

I guess local rebinding is not designed for refinements of classes
which have many clients like built-in classes. With local rebinding,
when we refine a built-in class such as String, we have to take care
carefully not to break existing code. This means that, it's
ridiculous to use the following refinement with local rebinding.

```
 module MathN
   refine Fixnum do
     def /(other) quo(other) end
   end
 end
```

My intention is to bring modularity into monkey patching, not to bring
extensibility. We already have enough extensibility in Ruby.

--
Shugo Maeda

=end

#76 - 12/06/2010 09:14 PM - headius (Charles Nutter)

=begin
On Sun, Dec 5, 2010 at 11:41 PM, Yehuda Katz wycats@gmail.com wrote:

> I think that, for this same reason, using should normally not apply
> outside of the immediate lexical scope. I actually believed that this was
> the default behavior, and explained why I thought it was a good idea on my
> blog (http://yehudakatz.com/2010/11/30/ruby-2-0-refinements-in-practice/).
> ...
> I can understand the utility of offering the inherited style when desired,
> but I don't think it should be the default. By default, the principle of
> this feature requires that refinements modify zero code outside its lexical
> scope.
> Perhaps a special form of using that would also affect subclasses (for
> example, for the case where Rails wants to expose ActiveSupport into its
> subclasses):
> class ActionController::Base
>   using ActiveSupport, inherited: true
> end

Im not sure inheritance is possible to support at all, since classes
don't know about their child classes (officially) and reopening a

class could not easily walk its children and make refinements apply to
them after the fact. Perhaps I missed the discussion about this
feature.

"using" should be applied similar to
"public/private/protected/module_function". In a sense, it's a
"refined" flag on the current scope that specifies that subsequent
scope creations should be "refined" in some way. There's no room for
leaking "private" and friends to definitions outside the lexical
scope, and the same should apply to "using".

By a similar token, "using" isn't impossible to apply after the fact
(i.e. adding a refinement "later" to some scope), but it is
infeasible. Because ideally we'd want to be able to flip a bit on
all calls about to be affected by the refinement, we'd need to
guarantee across all implementations that call sites are mutable at
runtime, where they aren't mutable now (or at least aren't
consistently and equivalently mutable). And just as mutating global
state would be bad (like flipping methods "private" in one thread
while they're being used in a another), so would having "using" change
the way a particular call site does its invocation after that site has
been constructed and released into the wild.

I suggest we drop the "pseudo" from "pseduolexical" in the Refinements
specification, and see where that takes us. Anything "pseudo" lexical
is dynamic scoping, which fails multiple performance, concurrency, and
obviousness tests for me.

- Charlie

=end

#77 - 12/06/2010 09:17 PM - headius (Charles Nutter)

=begin
On Sat, Dec 4, 2010 at 6:32 AM, Shugo Maeda shugo@ruby-lang.org wrote:

> > Note that this is just my opinion, and that I seem to be in the
> > minority ;-)
> > I hope that 1.9.x would be stable, but many other committers seem to
> > hope to include new feature in 1.9.x aggressively.
>
> I also hope that 1.9.x would be stable, but I'd like to develop
> aggressively in trunk. I don't think the current design and
> implementation of refinements are mature, but I can't make them mature
> on my own, so I'd like to have help from other committers.

This is what topic branches are for :) I don't believe refinements
should land on trunk, since it's not yet clear whether they should be
included at all. Landing them now, making multiple additional commits
to them, and propagating their changes throughout other
subsystems...all will make it harder to roll back Refinements if it is
decided they shouldn't get into standard Ruby.

I think everyone knows how to check out an alternative branch or
repository :) If they would like to assist you (and you would like
them to assist you) they can do it that way. It shouldn't be done on
mainline, in my opinion, until it has solidified a bit more and it's
certain to be included in an upcoming "standard" Ruby release.

- Charlie

=end

#78 - 12/06/2010 11:49 PM - shyouhei (Shyouhei Urabe)

=begin
(2010/12/06 21:17), Charles Oliver Nutter wrote:

> On Sat, Dec 4, 2010 at 6:32 AM, Shugo Maeda shugo@ruby-lang.org wrote:
>
> > > Note that this is just my opinion, and that I seem to be in the
> > > minority ;-)

> > > I hope that 1.9.x would be stable, but many other committers seem to
> > > hope to include new feature in 1.9.x aggressively.
>
> > I also hope that 1.9.x would be stable, but I'd like to develop
> > aggressively in trunk. I don't think the current design and
> > implementation of refinements are mature, but I can't make them mature
> > on my own, so I'd like to have help from other committers.
>
> This is what topic branches are for :)

No. That idea is not a SVN way. A trunk in SVN is an actively developed edge
branch. In contrast a master branch in Git is a merged collection of topic
branches, thus they are far less active than SVN trunk. So in SVN,
developments goes on the trunk.

> I don't believe refinements
> should land on trunk, since it's not yet clear whether they should be
> included at all.

I think it's all what you tell. You just don't like it, do you?

> Landing them now, making multiple additional commits
> to them, and propagating their changes throughout other
> subsystems...all will make it harder to roll back Refinements if it is
> decided they shouldn't get into standard Ruby.

You say "now" ... Do you believe it gets easier to roll back as time goes? I
don't think so. If it's a matter of time I can agree with you but...

> I think everyone knows how to check out an alternative branch or
> repository :) If they would like to assist you (and you would like
> them to assist you) they can do it that way.

Life is much easier if everyone assisted me testing our ruby_1_8_7 branch
every day. Did you know I'm planning a patchlevel release this month? Have
you compiled it and run tests? So far all who actually assisted me was Luis
Lavena only.

No, I'm not blaming you. It's all as usual. Assistance won't happen until
someone gets some trouble.

> It shouldn't be done on
> mainline, in my opinion, until it has solidified a bit more and it's
> certain to be included in an upcoming "standard" Ruby release.

Define your "a bit more". Otherwise you can reject a new feature as many time
as you want with this exact phrase.

=end

#79 - 12/07/2010 03:04 AM - lsegal (Loren Segal)

=begin
Hi,

On 12/6/2010 7:13 AM, Charles Oliver Nutter wrote:

> I suggest we drop the "pseudo" from "pseduolexical" in the Refinements
> specification, and see where that takes us. Anything "pseudo" lexical
> is dynamic scoping, which fails multiple performance, concurrency, and
> obviousness tests for me.

I would have to agree with Yehuda and Charlie on this. Refinements make
far more sense and are far easier to deal with when they are purely
lexical, both on a performance and a conceptual level. Although I'm not
nearly as qualified as Charlie to comment on the technical / performance
issues, it seems as though the entire point of refinements is to allow

scoped "monkey patching", and this can only work if there is
absolutely no scope leaking. Supporting inheritance gives way too much
wiggle room to unintentionally leak your scope that it effectively
limits how the feature can be used in real world situations.

I'd say that focusing on a pure lexical scoping for now would make it
far easier to get to a point where the implementation is "correct" and
can be commented on by more people in the community. Then, *if* the
performance / technical issues can be resolved, Yehuda's suggestion of
supporting opt-in inheritance-scoping through a hash argument would
still be on the table-- although a discussion should be had on whether
inheritance support really worth implementing after people are able to
test if a lexical-only implementation is sufficient.

- Loren

=end

#80 - 12/07/2010 03:23 AM - meta (mathew murphy)

=begin
On Sat, Dec 4, 2010 at 06:31, Charles Oliver Nutter headius@headius.comwrote:

> To be honest, I think it should be a using directive at the file's
> toplevel, so subsequently-parsed blocks know they're going to be
> refined.

Since I view refinements as a horrible feature I'll never use, I like the
idea of limiting the performance hit to only code that explicitly asks for
it at file level.

mathew

On Sat, Dec 4, 2010 at 06:31, Charles Oliver Nutter <headius@headius.com> wrote:
To be honest, I think it should be a using directive at the file's
toplevel, so subsequently-parsed blocks know they're going to be
refined.Since I view refinements as a horrible feature I'll never use, I like the idea of limiting the performance hit to only code that explicitly asks for it at file level.
mathew

=end

#81 - 12/07/2010 04:30 AM - wycats (Yehuda Katz)

=begin
Speaking as someone who writes a lot of libraries, I would be very likely to
use a purely lexical refinement feature all the time.

If the feature *was* limited to a purely lexical scope, most of my gems
would probably have a refinements module which I would "using" into the rest
of my code. This would allow me to get the benefit of the extra readability
of core class extensions without leaking those changes to other code.

On the other hand, if the feature leaked via local rebinding or via
inheritance, I would be much less likely (let's say, extremely unlikely) to
use the feature, instead being constantly paranoid that I could leak
refinements to code that didn't expect it (either by people subclassing my
classes or by leaking refinements to callees of my code not controlled by
me).

On the flip side, I would also be scared that people calling into my code
could leak refinements into my code, and probably take steps to protect my
code from that.

In short, this feature could either be really good (truly lexically scoped
"monkey patch protection") or really bad (conceptually as bad as it is now,
but harder to debug).

The difference is whether the feature, by default, never ever leaks
refinements into other scopes.

Yehuda Katz
Architect | Strobe

(ph) 718.877.1325

On Mon, Dec 6, 2010 at 1:23 PM, mathew meta@pobox.com wrote:

> On Sat, Dec 4, 2010 at 06:31, Charles Oliver Nutter headius@headius.comwrote:
>
> > To be honest, I think it should be a using directive at the file's
> > toplevel, so subsequently-parsed blocks know they're going to be
> > refined.
>
> Since I view refinements as a horrible feature I'll never use, I like the
> idea of limiting the performance hit to only code that explicitly asks for
> it at file level.
>
> mathew

Speaking as someone who writes a lot of libraries, I would be very likely to use a purely lexical refinement feature all the time.If the feature *was* limited to a purely lexical scope, most of my gems would probably have a refinements module which I would "using" into the rest of my code. This would allow me to get the benefit of the extra readability of core class extensions without leaking those changes to other code.
On the other hand, if the feature leaked via local rebinding or via inheritance, I would be much less likely (let's say, extremely unlikely) to use the feature, instead being constantly paranoid that I could leak refinements to code that didn't expect it (either by people subclassing my classes or by leaking refinements to callees of my code not controlled by me).
On the flip side, I would also be scared that people calling into my code could leak refinements into my code, and probably take steps to protect my code from that.In short, this feature could either be really good (truly lexically scoped "monkey patch protection") or really bad (conceptually as bad as it is now, but harder to debug).
The difference is whether the feature, by default, never ever leaks refinements into other scopes.Yehuda KatzArchitect | Strobe(ph) 718.877.1325
On Mon, Dec 6, 2010 at 1:23 PM, mathew <meta@pobox.com> wrote:
On Sat, Dec 4, 2010 at 06:31, Charles Oliver Nutter <headius@headius.com> wrote:

To be honest, I think it should be a using directive at the file's
toplevel, so subsequently-parsed blocks know they're going to be
refined.Since I view refinements as a horrible feature I'll never use, I like the idea of limiting the performance hit to only code that explicitly asks for it at file level.

mathew

=end

#82 - 12/07/2010 03:27 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/6 Yehuda Katz wycats@gmail.com:

> I think that, for this same reason, using should normally not apply
> outside of the immediate lexical scope. I actually believed that this was
> the default behavior, and explained why I thought it was a good idea on my
> blog (http://yehudakatz.com/2010/11/30/ruby-2-0-refinements-in-practice/).

I agree with you. I added inheritance support for frameworks such as
Rails, but it was wrong at least as the default behavior.

--
Shugo Maeda

=end

#83 - 12/07/2010 03:53 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/6 Charles Oliver Nutter headius@headius.com:

> > I also hope that 1.9.x would be stable, but I'd like to develop
> > aggressively in trunk. I don't think the current design and
> > implementation of refinements are mature, but I can't make them mature
> > on my own, so I'd like to have help from other committers.

> This is what topic branches are for :) I don't believe refinements
> should land on trunk, since it's not yet clear whether they should be
> included at all. Landing them now, making multiple additional commits
> to them, and propagating their changes throughout other
> subsystems...all will make it harder to roll back Refinements if it is
> decided they shouldn't get into standard Ruby.

What do `other subsystems' mean? libraries?
I think standard libraries should not use refinements until they are
stated as an official feature.

> I think everyone knows how to check out an alternative branch or
> repository :) If they would like to assist you (and you would like
> them to assist you) they can do it that way. It shouldn't be done on
> mainline, in my opinion, until it has solidified a bit more and it's
> certain to be included in an upcoming "standard" Ruby release.

As Shyouhei says, I doubt they can do it that way.

--
Shugo Maeda

=end

#84 - 12/07/2010 03:59 PM - headius (Charles Nutter)

=begin
On Mon, Dec 6, 2010 at 8:48 AM, Urabe Shyouhei shyouhei@ruby-lang.org wrote:

> (2010/12/06 21:17), Charles Oliver Nutter wrote:
>
>> This is what topic branches are for :)
>
> No. That idea is not a SVN way. A trunk in SVN is an actively developed edge
> branch. In contrast a master branch in Git is a merged collection of topic
> branches, thus they are far less active than SVN trunk. So in SVN,
> developments goes on the trunk.

My opinion, from dealing with SVN-based projects in the past:

Experimental development should never go on SVN trunk. Trunk should
only contain features that are intended to eventually get into a
release. I'm not sure Refinements qualifies (yet) as an official path
for Ruby, and so it should stay off trunk until that happens.

The alternative would be to let everyone doing any experiment
whatsoever commit to trunk, and then have to sort out what gets into
future releases and what does not. That does not make sense to me.

>> I don't believe refinements
>> should land on trunk, since it's not yet clear whether they should be
>> included at all.
>
> I think it's all what you tell. You just don't like it, do you?

I like Refinements, once they are "refined" to deal with the issues
brought up in this thread :) I have no objection to them going to MRI
trunk if they have reached the point where they're stable and
acceptable enough to officially be part of Ruby's future.

>> Landing them now, making multiple additional commits
>> to them, and propagating their changes throughout other
>> subsystems...all will make it harder to roll back Refinements if it is
>> decided they shouldn't get into standard Ruby.
>
> You say "now" ... Do you believe it gets easier to roll back as time goes? I
> don't think so. If it's a matter of time I can agree with you but...

It gets harder to roll back as time goes on. Maintaining experimental
features on a separate branch until they're blessed to become part of
Ruby's future would ensure they don't have to be reverted much later.

> > I think everyone knows how to check out an alternative branch or
> > repository :) If they would like to assist you (and you would like
> > them to assist you) they can do it that way.
>
> Life is much easier if everyone assisted me testing our ruby_1_8_7 branch
> every day. Did you know I'm planning a patchlevel release this month? Have
> you compiled it and run tests? So far all who actually assisted me was Luis
> Lavena only.
>
> No, I'm not blaming you. It's all as usual. Assistance won't happen until
> someone gets some trouble.

If I were more familiar with MRI codebase, I might try to help
"refine" Refinements, but unfortunately the best I can do is explain
my opinions and offer suggestions for improvement here on the list. I
can also try to implement the "refined" portions of Refinements in
JRuby.

As for others who *are* familiar with MRI...I don't see why they'd be
any more likely to assist in "refining" refinements if it were on
trunk versus on a branch. If they're interested in the feature,
they'll contribute wherever it is, no?

> > It shouldn't be done on
> > mainline, in my opinion, until it has solidified a bit more and it's
> > certain to be included in an upcoming "standard" Ruby release.
>
> Define your "a bit more". Otherwise you can reject a new feature as many time
> as you want with this exact phrase.

A bit more = once Matz says it should go on trunk (ideally, after
other implementers, ruby-core folks, and key community members agree
it's ready to go to trunk).

No offense is intended by either my commentary or my opinions. I just
would like to see Refinements land *after* these issues have been
sorted out. If there's more I can do to help that happen, please let
me know.

- Charlie

=end

#85 - 12/07/2010 04:15 PM - shugo (Shugo Maeda)

=begin
Hi,

2010/12/7 Charles Oliver Nutter headius@headius.com:

> > > It shouldn't be done on
> > > mainline, in my opinion, until it has solidified a bit more and it's
> > > certain to be included in an upcoming "standard" Ruby release.
> >
> > Define your "a bit more". Otherwise you can reject a new feature as many time
> > as you want with this exact phrase.
>
> A bit more = once Matz says it should go on trunk (ideally, after
> other implementers, ruby-core folks, and key community members agree
> it's ready to go to trunk).

I don't mean to merge refinements into trunk without Matz's permission.
I guess he's still wondering whether we need local rebinding.

--
Shugo Maeda

=end

#86 - 12/07/2010 05:04 PM - shyouhei (Shyouhei Urabe)

=begin
(2010/12/07 15:58), Charles Oliver Nutter wrote:

> On Mon, Dec 6, 2010 at 8:48 AM, Urabe Shyouhei shyouhei@ruby-lang.org wrote:
>
>> (2010/12/06 21:17), Charles Oliver Nutter wrote:
>>
>>> This is what topic branches are for :)
>>
>> No. That idea is not a SVN way. A trunk in SVN is an actively developed edge
>> branch. In contrast a master branch in Git is a merged collection of topic
>> branches, thus they are far less active than SVN trunk. So in SVN,
>> developments goes on the trunk.
>
> My opinion, from dealing with SVN-based projects in the past:
>
> Experimental development should never go on SVN trunk. Trunk should
> only contain features that are intended to eventually get into a
> release. I'm not sure Refinements qualifies (yet) as an official path
> for Ruby, and so it should stay off trunk until that happens.
>
> The alternative would be to let everyone doing any experiment
> whatsoever commit to trunk, and then have to sort out what gets into
> future releases and what does not. That does not make sense to me.

I think this is the whole point of this issue; what is a trunk. At least
until now, trunk has been what I explained. And you say it should not.

>>> I don't believe refinements
>>> should land on trunk, since it's not yet clear whether they should be
>>> included at all.
>>
>> I think it's all what you tell. You just don't like it, do you?
>
> I like Refinements, once they are "refined" to deal with the issues
> brought up in this thread :) I have no objection to them going to MRI
> trunk if they have reached the point where they're stable and
> acceptable enough to officially be part of Ruby's future.

I see. Sorry for the unnecessary attack.

>>> Landing them now, making multiple additional commits
>>> to them, and propagating their changes throughout other
>>> subsystems...all will make it harder to roll back Refinements if it is
>>> decided they shouldn't get into standard Ruby.
>>
>> You say "now" ... Do you believe it gets easier to roll back as time goes? I
>> don't think so. If it's a matter of time I can agree with you but...
>
> It gets harder to roll back as time goes on. Maintaining experimental
> features on a separate branch until they're blessed to become part of
> Ruby's future would ensure they don't have to be reverted much later.

So we have a consensus here. The difference is I'm expecting a revert so I'd
see it happen sooner to reduce the risk, while you don't want it at all.

=end

#87 - 12/07/2010 05:52 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:33610] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
on Tue, 7 Dec 2010 16:08:28 +0900, Shugo Maeda shugo@ruby-lang.org writes:

|I don't mean to merge refinements into trunk without Matz's permission.
|I guess he's still wondering whether we need local rebinding.

I do not object local rebinding (yet). In fact, I am slightly
negative against it. I am waiting for someone to try persuade me for
local rebinding, despite its potential performance penalty. No one
has ever tried yet.

```
                            matz.
```

=end

#88 - 12/07/2010 05:59 PM - rkh (Konstantin Haase)

=begin
Since I explained one use case I'd have for local rebinding: I think not having local rebinding is mostly what we want, local rebinding would mostly cause unwanted side effects and would be a pure horror to debug.

Konstantin

On Dec 7, 2010, at 09:52 , Yukihiro Matsumoto wrote:

> Hi,
>
> In message "Re: [ruby-core:33610] Re: [Ruby 1.9-Feature#4085][Open] Refinements and nested methods"
> on Tue, 7 Dec 2010 16:08:28 +0900, Shugo Maeda shugo@ruby-lang.org writes:
>
> |I don't mean to merge refinements into trunk without Matz's permission.
> |I guess he's still wondering whether we need local rebinding.
>
> I do not object local rebinding (yet). In fact, I am slightly
> negative against it. I am waiting for someone to try persuade me for
> local rebinding, despite its potential performance penalty. No one
> has ever tried yet.
>
> ```
>                             matz.
> ```

=end

#89 - 12/07/2010 07:34 PM - headius (Charles Nutter)

=begin
On Tue, Dec 7, 2010 at 2:58 AM, Haase, Konstantin
Konstantin.Haase@student.hpi.uni-potsdam.de wrote:

> Since I explained one use case I'd have for local rebinding: I think not having local rebinding is mostly what we want, local rebinding would mostly cause unwanted side effects and would be a pure horror to debug.

So to clarify (for me) what you are saying: you had a use case for
local rebinding, but you don't think it's worth the side effects, yes?
In other words, a +1 reverted to a 0 or -1?

- Charlie

=end

#90 - 12/07/2010 07:57 PM - rkh (Konstantin Haase)

=begin

On Dec 7, 2010, at 11:34 , Charles Oliver Nutter wrote:

> On Tue, Dec 7, 2010 at 2:58 AM, Haase, Konstantin
> Konstantin.Haase@student.hpi.uni-potsdam.de wrote:
>
>> Since I explained one use case I'd have for local rebinding: I think not having local rebinding is mostly what we want, local rebinding would

> > mostly cause unwanted side effects and would be a pure horror to debug.
>
> So to clarify (for me) what you are saying: you had a use case for
> local rebinding, but you don't think it's worth the side effects, yes?
> In other words, a +1 reverted to a 0 or -1?
>
> - Charlie

Yes. Even ignoring the performance impact, the benefits of local rebinding don't justify the harm it could do on an architectural level.
The only option would be introducing a switch (as proposed for inheritance), but that would get us back to a global performance hit, would it?

One other thing:

At the moment, refinements are not only activated by using, but by refine, too. Wouldn't that mean on one hand, that refine would have to be a keyword too, and on the other hand, that you cannot import refinements at all, if those are purely lexically scoped?

Konstantin

=end

#91 - 06/28/2011 06:18 AM - nahi (Hiroshi Nakamura)

- Target version changed from 1.9.3 to 2.0.0

#92 - 03/18/2012 06:42 PM - nahi (Hiroshi Nakamura)

- Assignee set to shugo (Shugo Maeda)

#93 - 03/18/2012 06:46 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

#94 - 07/23/2012 11:00 AM - shugo (Shugo Maeda)

- Assignee changed from shugo (Shugo Maeda) to ko1 (Koichi Sasada)

Matz said the only problem is performance before, so I assign this issue to ko1.

#95 - 08/02/2012 08:41 PM - shugo (Shugo Maeda)

shugo (Shugo Maeda) wrote:

> Matz said the only problem is performance before, so I assign this issue to ko1.

I've committed refinements in r36596 with Matz's permission. However,
it's just an experimental feature, and may be reverted before the release of 2.0.
Please try it, and give us feedback.

#96 - 08/12/2012 03:12 AM - trans (Thomas Sawyer)

=begin
One question I have is how this would work with something like Facets, including its ability to cherry pick methods.

So lets say we have facets/object/foo and facets/object/bar.

facets/object/foo.rb
module Facets
refine Object
def foo
...
end
end
end

lib/facets/object/bar.rb
module Facets
refine Object
def bar
...
end
end
end

So now one could include "Facets" in their application's namespace?

require 'facets/object/foo'
require 'facets/object/bar'

module MyApp
include Facets
end

Would that work?
=end

#97 - 08/12/2012 04:04 AM - Eregon (Benoit Daloze)

trans (Thomas Sawyer) wrote:

> =begin
> module MyApp
> include Facets
> end
>
> Would that work?
> =end

Yes, except you need to use using and not include in module MyApp.

#98 - 09/22/2012 09:05 AM - ko1 (Koichi Sasada)

- Assignee changed from ko1 (Koichi Sasada) to shugo (Shugo Maeda)

Any problem now?

#99 - 10/28/2012 12:24 PM - shugo (Shugo Maeda)

- Assignee changed from shugo (Shugo Maeda) to matz (Yukihiro Matsumoto)

ko1 (Koichi Sasada) wrote:

> Any problem now?

I like the current behavior of Refinements, so please give me feedback if you don't like it.

Some objections to Refinements are summarized below:

- Refinements are too complex.
- Refinements should support local rebinding.
- using should be a keyword.
- refine should be a keyword.

Matz should decide whether Refinements should be included in Ruby 2.0, but I'm not sure whether he has tried Refinements or not.

#100 - 11/02/2012 07:03 AM - headius (Charles Nutter)

I would like to request time to implement refinements as they stand today in JRuby.

We have not had much time to look into how refinements will affect the way we optimize Ruby code, structure class hierarchies, and so on. The impact could be severe, or it could be minimal. I am worried because I don't know, and because up until today (Matz's keynote) I thought they were still questionable for 2.0.0.

So...we will implement refinements in JRuby (possibly on a branch) as soon as possible...within the next month for sure. I hope if we run into issues with the currently-specified behavior there will be time to make changes.

#101 - 11/02/2012 11:54 AM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> I would like to request time to implement refinements as they stand today in JRuby.
>
> We have not had much time to look into how refinements will affect the way we optimize Ruby code, structure class hierarchies, and so on. The impact could be severe, or it could be minimal. I am worried because I don't know, and because up until today (Matz's keynote) I thought they were still questionable for 2.0.0.

> So...we will implement refinements in JRuby (possibly on a branch) as soon as possible...within the next month for sure. I hope if we run into issues with the currently-specified behavior there will be time to make changes.

Thanks for your consideration of Refinements.
Please inform me if the current behavior of Refinements is hard to implement in JRuby.

#102 - 11/03/2012 02:06 AM - headius (Charles Nutter)

Ok, early notes from writing up specs and starting to explore an implementation for JRuby.

- Refinements in modules and class hierarchies does not seem like a problem to me yet.
- Refinements are "used" in temporal order...methods added before "using" won't see refinements, refinements added after "using" won't be applied. I think this is a good thing, since it allows us to have a one-shot flag for refinements on methods at definition time.
- I notice 2.0.0 does not appear to have a performance impact from refinements in a simple case. No idea how refinements interact with cache invalidation yet.

Now, the bad:

- Months ago when the original proposal came out, I expressed my concern about refinements applying to module_eval and friends. I still strongly object to this behavior.

The problems I enumerated then still apply. Because module/class_eval can take any arbitrary block, including blocks of code from distant locations that do not know anything about used refinements, all calls in all blocks would need to check for refinements every time. This is the simple performance/optimization concern. MRI's invalidation mechanisms are simple enough that it may not affect MRI right now...but if invalidation mechanisms like those in JRuby are ever desired in MRI, I don't think this feature can exist.

It is also a GROSS security issue on the scale of block-as-binding. If a library can force code within a block to make completely different calls than the author intended, you're going to cause terrible confusion. Say I write some code in a block and pass it to a badly-written or malicious receiver. That receiver module_eval's my block in a refined scope, changing one or more of the calls I wrote. Suddenly my code, tested in my environment, can behave completely differently just by being passed as a block. If there are errors, they'll be errors reporting my code calling some other method I never intended.

Look at this code and tell me what method it calls:

def my_code(string)
your_code { string.to_i(16) }
end

The answer is that you can't tell me what method it is calling because the your_code method can basically force it to call anything.

This is actually *worse* than monkey patching, because after the call to your_code has completed, I have no evidence that a refinement was active. I need to go spelunking in your_code to figure out why my code didn't behave the way it should.

This is dynamic application of refinements, which has been hotly debated and which I *thought* was supposed to be removed. I assume it has been left in because it is required to apply refinements "magically" to all-block code like rspec. I do not see this as an excuse to introduce such an unpredictable feature.

Thoughts?

I am adding rubyspecs and continuing my research.

#103 - 11/03/2012 02:12 AM - headius (Charles Nutter)

Adding a couple additional positive notes:

- For now, I don't see a reason for using and refine to be keywords.

using and refine basically just add some additional state to modules and methods.

For refine:

- add to the module a mapping from refined class/module to the refinements that should be applied
- mark the containing module as having refinements active

For using:

- copy to the target module all currently active module => refinement mappings.
- tag methods defined after the using call to indicate that they should use refinement lookup

I think this is all doable without major alteration of most VMs caching, jitting, optimization mechanisms.

module_eval and class_eval are a different issue. There's no indication ahead of time that calls within those scopes need to use refinements, so you have to check constantly and enlist in refinement lookup unconditionally.

#104 - 11/03/2012 10:12 AM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> Ok, early notes from writing up specs and starting to explore an implementation for JRuby.

Thanks.

> - Refinements in modules and class hierarchies does not seem like a problem to me yet.
> - Refinements are "used" in temporal order...methods added before "using" won't see refinements, refinements added after "using" won't be applied. I think this is a good thing, since it allows us to have a one-shot flag for refinements on methods at definition time.

The current behavior is mainly for an implementation reason, but Matz and ko1 seem not to like it:(

> - Months ago when the original proposal came out, I expressed my concern about refinements applying to module_eval and friends. I still strongly object to this behavior.

I also wonder whether module_eval with blocks should be affected by refinements or not, but I think module_eval with strings (e.g., M.module_eval("C.new.foo")) has no problem, right?

> This is dynamic application of refinements, which has been hotly debated and which I *thought* was supposed to be removed. I assume it has been left in because it is required to apply refinements "magically" to all-block code like rspec. I do not see this as an excuse to introduce such an unpredictable feature.

instance_eval and module_eval themselves have the same problem because they change self "magically".
At first, I thought they are evil, but they are popular now.
I'd like to ask Matz's opinion.

#105 - 11/04/2012 01:53 AM - headius (Charles Nutter)

On Fri, Nov 2, 2012 at 7:12 PM, shugo (Shugo Maeda)
redmine@ruby-lang.org wrote:

> headius (Charles Nutter) wrote:
>
> > - Refinements in modules and class hierarchies does not seem like a problem to me yet.
> > - Refinements are "used" in temporal order...methods added before "using" won't see refinements, refinements added after "using" won't be applied. I think this is a good thing, since it allows us to have a one-shot flag for refinements on methods at definition time.
>
> The current behavior is mainly for an implementation reason, but Matz and ko1 seem not to like it:(

I commented on ko1's bug. I see using a bit like visibility changes,
only affecting methods defined later on.

I understand the implementation reason as well...in order to limit the
damage of refinements, your impl flags methods as having refinements
active. For optimization purposes, that means we know at definition
time whether we need to handle refinements at all.

> > - Months ago when the original proposal came out, I expressed my concern about refinements applying to module_eval and friends. I still strongly object to this behavior.
>
> I also wonder whether module_eval with blocks should be affected by refinements or not, but I think module_eval with strings (e.g., M.module_eval("C.new.foo")) has no problem, right?

String eval would not be a problem, that is correct. We would be able
to see at eval time that the target module has refinements active.

> > This is dynamic application of refinements, which has been hotly debated and which I *thought* was supposed to be removed. I assume it has been left in because it is required to apply refinements "magically" to all-block code like rspec. I do not see this as an excuse to introduce such an unpredictable feature.
>
> instance_eval and module_eval themselves have the same problem because they change self "magically".
> At first, I thought they are evil, but they are popular now.
> I'd like to ask Matz's opinion.

Yes, module_eval, class_eval, and instance_eval are all problematic
because of the self changing, but module_eval and class_eval are
especially bad if they force refinements on code that doesn't know
about them.

I am starting to see some intractable problems with refinements, unfortunately.

In order to avoid having every call in the system check for
refinements, they are applied in evaluation order. However, this means
that the load order of scripts can now completely change which methods
get called. For example...

a.rb:

```
class Foo
def go(obj)
obj.something
end
end
```

b.rb:

```
class Foo
using Baz # refines the "something" call
end
```

If the files are loaded in the order a, b, no refinements are applied
to the something call. If b is loaded before a, refinements are
applied to the something call. No other features in Ruby are so
sensitive to load order (other than those that introspect classes and
methods, obviously).

The alternative is to have refinements not be applied temporally.
However this means every call in the system needs to check for
refinements every time. Given the complexity of method lookup in Ruby
today, adding refinements to that process seems like a terrible idea.

In order to cache a method call in the presence of refinements, we
need to track all of the following:

1. whether any refinements are active
2. whether there are refinements that affect the class of the target of the method call
3. whether refinements that affect the target class redefine the target method

If any of these change at any time, we need to invalidate the cache.
This is on top of all the information we need to do to cache methods
normally.

At this point I would not vote for refinements to be included in Ruby
2.0. I feel like there are far too many edge cases and implementation
concerns.

I will continue my investigation.

#106 - 11/04/2012 01:53 AM - headius (Charles Nutter)

More thoughts...

I could get behind refinements if using were a keyword (so we could
tell at parse time refinements would be active) and purely lexical.
The following features of the current implementation would have to be
removed:

- refinements cascading down class hierarchies
- refinements affecting code in module_eval'ed blocks

If using became a keyword and purely lexical, the following examples
would be fine:

rails_thing.rb:

```
class Foo
using ActiveRecord::SomeExt
...
```

end

rspec_thing.rb:

using RSpec

describe 'Refinements' do
it 'should be purely lexical' do
...

If refinements can affect code outside the lexical scope where they
are activated, I believe it will be confusing, potentially dangerous,
very hard to debug, and potentially difficult or impossible to
implement without slowing all of Ruby down.

On Sat, Nov 3, 2012 at 10:38 AM, Charles Oliver Nutter
headius@headius.com wrote:

> On Fri, Nov 2, 2012 at 7:12 PM, shugo (Shugo Maeda)
> redmine@ruby-lang.org wrote:
>
>> headius (Charles Nutter) wrote:
>>
>>> - Refinements in modules and class hierarchies does not seem like a problem to me yet.
>>> - Refinements are "used" in temporal order...methods added before "using" won't see refinements, refinements added after "using" won't be applied. I think this is a good thing, since it allows us to have a one-shot flag for refinements on methods at definition time.
>>
>> The current behavior is mainly for an implementation reason, but Matz and ko1 seem not to like it:(
>
> I commented on ko1's bug. I see using a bit like visibility changes,
> only affecting methods defined later on.
>
> I understand the implementation reason as well...in order to limit the
> damage of refinements, your impl flags methods as having refinements
> active. For optimization purposes, that means we know at definition
> time whether we need to handle refinements at all.
>
>>> - Months ago when the original proposal came out, I expressed my concern about refinements applying to module_eval and friends. I still strongly object to this behavior.
>>
>> I also wonder whether module_eval with blocks should be affected by refinements or not, but I think module_eval with strings (e.g., M.module_eval("C.new.foo")) has no problem, right?
>
> String eval would not be a problem, that is correct. We would be able
> to see at eval time that the target module has refinements active.
>
>>> This is dynamic application of refinements, which has been hotly debated and which I *thought* was supposed to be removed. I assume it has been left in because it is required to apply refinements "magically" to all-block code like rspec. I do not see this as an excuse to introduce such an unpredictable feature.
>>
>> instance_eval and module_eval themselves have the same problem because they change self "magically".
>> At first, I thought they are evil, but they are popular now.
>> I'd like to ask Matz's opinion.
>
> Yes, module_eval, class_eval, and instance_eval are all problematic
> because of the self changing, but module_eval and class_eval are
> especially bad if they force refinements on code that doesn't know
> about them.
>
> I am starting to see some intractable problems with refinements, unfortunately.
>
> In order to avoid having every call in the system check for
> refinements, they are applied in evaluation order. However, this means
> that the load order of scripts can now completely change which methods
> get called. For example...
>
> a.rb:

```
class Foo
def go(obj)
obj.something
end
end
```

b.rb:

```
class Foo
using Baz # refines the "something" call
end
```

If the files are loaded in the order a, b, no refinements are applied
to the something call. If b is loaded before a, refinements are
applied to the something call. No other features in Ruby are so
sensitive to load order (other than those that introspect classes and
methods, obviously).

The alternative is to have refinements not be applied temporally.
However this means every call in the system needs to check for
refinements every time. Given the complexity of method lookup in Ruby
today, adding refinements to that process seems like a terrible idea.

In order to cache a method call in the presence of refinements, we
need to track all of the following:

1. whether any refinements are active
2. whether there are refinements that affect the class of the target of the method call
3. whether refinements that affect the target class redefine the target method

If any of these change at any time, we need to invalidate the cache.
This is on top of all the information we need to do to cache methods
normally.

At this point I would not vote for refinements to be included in Ruby
2.0. I feel like there are far too many edge cases and implementation
concerns.

I will continue my investigation.

#107 - 11/04/2012 05:33 AM - headius (Charles Nutter)

Discussed refinements a bit with Matz + ko1 plus a_matsuda and others at lunch...here's summary.

The using method currently touches cref, making it largely scoped like constants. If refinement lookup at a call site proceeds the same way, it would simplify things in my head quite a bit.

Benefits to having refinement lookup follow cref:

- No frame field is necessary
- Easier-to-understand behavior, since the lookup would mimic constant lookup (lexical then hierarchical)

There are down sides:

- module_eval cases would not be possible, since like constant lookup refinements cannot be injected into a scope after the fact
- It is not purely lexical, as I had hoped. However, I recognize that having to "using" in every scope where you want refinements to be active would be painful.
- It may need a global invalidation guard, like constants. It may not, though, since refinements are applied only once, at definition time; so after caching once we may have everything we need for future invalidation.

One concern of mine was avoiding refinement checks at all call sites. With cref-based "using" there's no way to determine at parse time which methods might need refinements. We can determine it at definition time, but the AST is already parsed at that point, so we'd have to rewrite it to have "refined calls" instead of normal calls, or something similar. This also impacts JRuby's ahead-of-time compilation support...we do not have the luxury of running the target code for precompilation, so we can't see if there are refinements active.

For normal runtime jitted code, rewriting the AST or adding a "refined" flag to call nodes may be feasible. Calls can then use a refined call site only in the cases where refinements are active.

For AOT mode, I think the best I can do is to have a flag on the call sites to indicate if refinements are active. If there's no refinements for the first call, there will be no refinements for any future calls, and we can turn that logic off. It would still require that boolean check every time...I don't see a way to eliminate that. (Side note: on invokedynamic, that boolean check would end up free, so it may not be a huge deal long term).

I will attempt to prototype pure cref-based refinement lookup over the next week.

#108 - 11/04/2012 05:59 AM - ko1 (Koichi Sasada)

I have another implementation idea, using "prepend".
I also try it next week.
(I can't explain this idea in English, so I will write code).

(2012/11/03 14:33), headius (Charles Nutter) wrote:

> Issue #4085 has been updated by headius (Charles Nutter).
>
> Discussed refinements a bit with Matz + ko1 plus a_matsuda and others at lunch...here's summary.
>
> The using method currently touches cref, making it largely scoped like constants. If refinement lookup at a call site proceeds the same way, it would simplify things in my head quite a bit.
>
> Benefits to having refinement lookup follow cref:
>
> - No frame field is necessary
> - Easier-to-understand behavior, since the lookup would mimic constant lookup (lexical then hierarchical)
>
> There are down sides:
>
> - module_eval cases would not be possible, since like constant lookup refinements cannot be injected into a scope after the fact
> - It is not purely lexical, as I had hoped. However, I recognize that having to "using" in every scope where you want refinements to be active would be painful.
> - It may need a global invalidation guard, like constants. It may not, though, since refinements are applied only once, at definition time; so after caching once we may have everything we need for future invalidation.
>
> One concern of mine was avoiding refinement checks at all call sites. With cref-based "using" there's no way to determine at parse time which methods might need refinements. We can determine it at definition time, but the AST is already parsed at that point, so we'd have to rewrite it to have "refined calls" instead of normal calls, or something similar. This also impacts JRuby's ahead-of-time compilation support...we do not have the luxury of running the target code for precompilation, so we can't see if there are refinements active.
>
> For normal runtime jitted code, rewriting the AST or adding a "refined" flag to call nodes may be feasible. Calls can then use a refined call site only in the cases where refinements are active.
>
> For AOT mode, I think the best I can do is to have a flag on the call sites to indicate if refinements are active. If there's no refinements for the first call, there will be no refinements for any future calls, and we can turn that logic off. It would still require that boolean check every time...I don't see a way to eliminate that. (Side note: on invokedynamic, that boolean check would end up free, so it may not be a huge deal long term).
>
> ## I will attempt to prototype pure cref-based refinement lookup over the next week.
>
> Feature #4085: Refinements and nested methods
> https://bugs.ruby-lang.org/issues/4085#change-32309
>
> Author: shugo (Shugo Maeda)
> Status: Assigned
> Priority: Normal
> Assignee: matz (Yukihiro Matsumoto)
> Category: core
> Target version: 2.0.0
>
> =begin
> As I said at RubyConf 2010, I'd like to propose a new features called
> "Refinements."
>
> Refinements are similar to Classboxes. However, Refinements doesn't
> support local rebinding as mentioned later. In this sense,
> Refinements might be more similar to selector namespaces, but I'm not
> sure because I have never seen any implementation of selector
> namespaces.
>
> In Refinements, a Ruby module is used as a namespace (or classbox) for
> class extensions. Such class extensions are called refinements. For
> example, the following module refines Fixnum.
>
> module MathN
> refine Fixnum do
> def /(other) quo(other) end
> end
> end
>
> Module#refine(klass) takes one argument, which is a class to be
> extended. Module#refine also takes a block, where additional or
> overriding methods of klass can be defined. In this example, MathN

refines Fixnum so that 1 / 2 returns a rational number (1/2) instead
of an integer 0.

This refinement can be enabled by the method using.

class Foo
using MathN

```
def foo
  p 1 / 2
end
```

end

f = Foo.new
f.foo #=> (1/2)
p 1 / 2

In this example, the refinement in MathN is enabled in the definition
of Foo. The effective scope of the refinement is the innermost class,
module, or method where using is called; however the refinement is not
enabled before the call of using. If there is no such class, module,
or method, then the effective scope is the file where using is called.
Note that refinements are pseudo-lexically scoped. For example,
foo.baz prints not "FooExt#bar" but "Foo#bar" in the following code:

class Foo
def bar
puts "Foo#bar"
end

```
def baz
  bar
end
```

end

module FooExt
refine Foo do
def bar
puts "FooExt#bar"
end
end
end

module Quux
using FooExt

```
foo = Foo.new
foo.bar  # => FooExt#bar
foo.baz  # => Foo#bar
```

end

Refinements are also enabled in reopened definitions of classes using
refinements and definitions of their subclasses, so they are
pseudo-lexically scoped.

class Foo
using MathN
end

class Foo
MathN is enabled in a reopened definition.
p 1 / 2 #=> (1/2)
end

class Bar < Foo
MathN is enabled in a subclass definition.
p 1 / 2 #=> (1/2)
end

If a module or class is using refinements, they are enabled in
module_eval, class_eval, and instance_eval if the receiver is the
class or module, or an instance of the class.

```
module A
using MathN
end
class B
using MathN
end
MathN.module_eval do
p 1 / 2  #=> (1/2)
end
A.module_eval do
p 1 / 2  #=> (1/2)
end
B.class_eval do
p 1 / 2  #=> (1/2)
end
B.new.instance_eval do
p 1 / 2  #=> (1/2)
end
```

Besides refinements, I'd like to propose new behavior of nested methods.
Currently, the scope of a nested method is not closed in the outer method.

```
def foo
def bar
puts "bar"
end
bar
end
foo  #=> bar
bar  #=> bar
```

In Ruby, there are no functions, but only methods. So there are no
right places where nested methods are defined. However, if
refinements are introduced, a refinement enabled only in the outer
method would be the right place. For example, the above code is
almost equivalent to the following code:

```
def foo
klass = self.class
m = Module.new {
refine klass do
def bar
puts "bar"
end
end
}
using m
bar
end
foo  #=> bar
bar  #=> NoMethodError
```

The attached patch is based on SVN trunk r29837.
=end

--
// SASADA Koichi at atdot dot net

#109 - 11/04/2012 08:53 AM - matsuda (Akira Matsuda)

Playing with refinements after talking with Charlie, I found
refinements are not very much attractive feature without module_eval.
I think I do understand the down sides Charlie described.
And I'm sure people will abuse this refinements + module_eval
technique if they are allowed to, because it just looks cool.

So, I guess the point is whether to allow users to write code like this or not;

```
module A
refine Fixnum do
def +(o); self * o; end
end
end
```

class C
def foo(&b)
Module.new { using A }.module_eval &b
end
end

C.new.foo { p 2 + 3 }
#=> 6

My personal opinion is, still, as a library / framework author, I'm
sort of anticipating to see such future that people are abusing this.
People may find good practices to control the power somehow, then it
might definitely push the DSL culture up to the absolute next level,
which must be totally exciting.

On Sat, Nov 3, 2012 at 2:57 PM, SASADA Koichi ko1@atdot.net wrote:

> I have another implementation idea, using "prepend".
> I also try it next week.
> (I can't explain this idea in English, so I will write code).
>
> (2012/11/03 14:33), headius (Charles Nutter) wrote:
>
> > Issue #4085 has been updated by headius (Charles Nutter).
> >
> > Discussed refinements a bit with Matz + ko1 plus a_matsuda and others at lunch...here's summary.
> >
> > The using method currently touches cref, making it largely scoped like constants. If refinement lookup at a call site proceeds the same way, it would simplify things in my head quite a bit.
> >
> > Benefits to having refinement lookup follow cref:
> >
> > - No frame field is necessary
> > - Easier-to-understand behavior, since the lookup would mimic constant lookup (lexical then hierarchical)
> >
> > There are down sides:
> >
> > - module_eval cases would not be possible, since like constant lookup refinements cannot be injected into a scope after the fact
> > - It is not purely lexical, as I had hoped. However, I recognize that having to "using" in every scope where you want refinements to be active would be painful.
> > - It may need a global invalidation guard, like constants. It may not, though, since refinements are applied only once, at definition time; so after caching once we may have everything we need for future invalidation.
> >
> > One concern of mine was avoiding refinement checks at all call sites. With cref-based "using" there's no way to determine at parse time which methods might need refinements. We can determine it at definition time, but the AST is already parsed at that point, so we'd have to rewrite it to have "refined calls" instead of normal calls, or something similar. This also impacts JRuby's ahead-of-time compilation support...we do not have the luxury of running the target code for precompilation, so we can't see if there are refinements active.
> >
> > For normal runtime jitted code, rewriting the AST or adding a "refined" flag to call nodes may be feasible. Calls can then use a refined call site only in the cases where refinements are active.
> >
> > For AOT mode, I think the best I can do is to have a flag on the call sites to indicate if refinements are active. If there's no refinements for the first call, there will be no refinements for any future calls, and we can turn that logic off. It would still require that boolean check every time...I don't see a way to eliminate that. (Side note: on invokedynamic, that boolean check would end up free, so it may not be a huge deal long term).
> >
> > ## I will attempt to prototype pure cref-based refinement lookup over the next week.
> >
> > Feature #4085: Refinements and nested methods
> > https://bugs.ruby-lang.org/issues/4085#change-32309
> >
> > Author: shugo (Shugo Maeda)
> > Status: Assigned
> > Priority: Normal
> > Assignee: matz (Yukihiro Matsumoto)
> > Category: core
> > Target version: 2.0.0
> >
> > =begin
> > As I said at RubyConf 2010, I'd like to propose a new features called
> > "Refinements."
> >
> > Refinements are similar to Classboxes. However, Refinements doesn't
> > support local rebinding as mentioned later. In this sense,
> > Refinements might be more similar to selector namespaces, but I'm not

sure because I have never seen any implementation of selector namespaces.

In Refinements, a Ruby module is used as a namespace (or classbox) for class extensions. Such class extensions are called refinements. For example, the following module refines Fixnum.

module MathN
refine Fixnum do
def /(other) quo(other) end
end
end

Module#refine(klass) takes one argument, which is a class to be extended. Module#refine also takes a block, where additional or overriding methods of klass can be defined. In this example, MathN refines Fixnum so that 1 / 2 returns a rational number (1/2) instead of an integer 0.

This refinement can be enabled by the method using.

class Foo
using MathN

```
def foo
  p 1 / 2
end
```

end

f = Foo.new
f.foo #=> (1/2)
p 1 / 2

In this example, the refinement in MathN is enabled in the definition of Foo. The effective scope of the refinement is the innermost class, module, or method where using is called; however the refinement is not enabled before the call of using. If there is no such class, module, or method, then the effective scope is the file where using is called. Note that refinements are pseudo-lexically scoped. For example, foo.baz prints not "FooExt#bar" but "Foo#bar" in the following code:

class Foo
def bar
puts "Foo#bar"
end

```
def baz
  bar
end
```

end

module FooExt
refine Foo do
def bar
puts "FooExt#bar"
end
end
end

module Quux
using FooExt

```
foo = Foo.new
foo.bar  # => FooExt#bar
foo.baz  # => Foo#bar
```

end

Refinements are also enabled in reopened definitions of classes using refinements and definitions of their subclasses, so they are *pseudo*-lexically scoped.

class Foo

```
using MathN
end

class Foo
# MathN is enabled in a reopened definition.
p 1 / 2  #=> (1/2)
end

class Bar < Foo
# MathN is enabled in a subclass definition.
p 1 / 2  #=> (1/2)
end
```

If a module or class is using refinements, they are enabled in
module_eval, class_eval, and instance_eval if the receiver is the
class or module, or an instance of the class.

```
module A
using MathN
end
class B
using MathN
end
MathN.module_eval do
p 1 / 2  #=> (1/2)
end
A.module_eval do
p 1 / 2  #=> (1/2)
end
B.class_eval do
p 1 / 2  #=> (1/2)
end
B.new.instance_eval do
p 1 / 2  #=> (1/2)
end
```

Besides refinements, I'd like to propose new behavior of nested methods.
Currently, the scope of a nested method is not closed in the outer method.

```
def foo
def bar
puts "bar"
end
bar
end
foo  #=> bar
bar  #=> bar
```

In Ruby, there are no functions, but only methods. So there are no
right places where nested methods are defined. However, if
refinements are introduced, a refinement enabled only in the outer
method would be the right place. For example, the above code is
almost equivalent to the following code:

```
def foo
klass = self.class
m = Module.new {
refine klass do
def bar
puts "bar"
end
end
}
using m
bar
end
foo  #=> bar
bar  #=> NoMethodError
```

The attached patch is based on SVN trunk r29837.
=end

--
// SASADA Koichi at atdot dot net

--
Akira Matsudaronnie@dio.jp

#110 - 11/05/2012 07:25 AM - Anonymous

Hi,

In message "Re: [ruby-core:48828] Re: [ruby-trunk - Feature #4085] Refinements and nested methods"
on Sun, 4 Nov 2012 08:42:11 +0900, Akira Matsuda ronnie@dio.jp writes:
|
|Playing with refinements after talking with Charlie, I found
|refinements are not very much attractive feature without module_eval.
|I think I do understand the down sides Charlie described.
|And I'm sure people will abuse this refinements + module_eval
|technique if they are allowed to, because it just looks cool.

I agree. Even though there's chance to make program behavior
unpredictable, extending behavior in block with refinements would
enhance the possibility of expressiveness. ActiveRecord example is
one of them. As far as I understand, Charlie came up with an idea to
resolve refinements at the first method invocation. If the idea is a
right one, we can allow such cases.

But I am not sure allowing refinement application in blocks should be
done by module_eval or not. Generating anonymous module for every
block execution sound inefficient.

```
                            matz.
```

#111 - 11/05/2012 11:44 AM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> Discussed refinements a bit with Matz + ko1 plus a_matsuda and others at lunch...here's summary.

Thanks for the summary.

The current implementation is cref-based because I came up with the idea inspired by pseudo lexical scope of constants. However, I admit that the current implementation has some issues, and don't stick to it.

I'd like to see your implementation and ko1's prepend-based implementation.

#112 - 11/05/2012 11:50 AM - shugo (Shugo Maeda)

matz wrote:

> But I am not sure allowing refinement application in blocks should be
> done by module_eval or not. Generating anonymous module for every
> block execution sound inefficient.

Such an anonymous module may be reused in a_matsuda's cases, but method caching may be a worse problem. In the current implementation, using invalidates the global method cache and module_eval invokes using internally, which means that the global method cache is invalidated for each call of module_eval.

#113 - 11/05/2012 01:33 PM - headius (Charles Nutter)

shugo (Shugo Maeda) wrote:

> matz wrote:
>
> > But I am not sure allowing refinement application in blocks should be
> > done by module_eval or not. Generating anonymous module for every
> > block execution sound inefficient.
>
> Such an anonymous module may be reused in a_matsuda's cases, but method caching may be a worse problem. In the current implementation, using invalidates the global method cache and module_eval invokes using internally, which means that the global method cache is invalidated for each call of module_eval.

To me, this means the module_eval feature absolutely cannot be supported unless a new implementation can be found. No normal feature of Ruby should force full cache invalidation globally.

#114 - 11/05/2012 01:39 PM - matz (Yukihiro Matsumoto)

I agree with Charles here. Avoiding global cache invalidation is priority one.
At the same time, I want a way to enable refinement to given block.

Matz.

#115 - 11/05/2012 01:53 PM - headius (Charles Nutter)

Sorry, this got double-posted because I thought my email did not come through properly.

On Sun, Nov 4, 2012 at 8:50 PM, shugo (Shugo Maeda)
redmine@ruby-lang.org wrote:

> Such an anonymous module may be reused in a_matsuda's cases, but method caching may be a worse problem. In the current implementation, using invalidates the global method cache and module_eval invokes using internally, which means that the global method cache is invalidated for each call of module_eval.

Unless an implementation can be found that doesn't do this, I don't
see how the module_eval feature can be supported. It seems totally
unreasonable to me that the global method cache would be invalidated
in this way.

#116 - 11/05/2012 02:17 PM - headius (Charles Nutter)

I currently don't see any way to support enabling refinements in blocks without a global, unavoidable impact to all call sites.

If refinements are applied lexically, then a block suddenly having refinements active does not make sense, since the code in the block is not lexically surrounded by a scope with active refinements.

If refinements are applied based on the cref module, then a block still can't see them because cref for the block is determined lexically. This was explicitly done to allow constant lookup to always proceed based on lexical scopes.

Formally, the clearest way for refinements to work is for you to explicitly opt into them everywhere you want to use them. A couple thoughts about different forms:

1. using as a scope-opening keyword like "class" or "module":

using RSpec
describe "Some Spec" do
it "does some RSpec things" do # "it" comes from refinements
'foo'.should == 'foo' # "should" comes from refinements
end
end
end

This is clear to the user and the parser that all calls downstream from the "using" must consider refinements. They would see the refinements based on cref (or something similar) because of the explicit nesting of scopes.

1. "using" as a pseudo-keyword, affecting lexical scopes and code following "using"

using RSpec

describe "Some Spec" do
it "does some RSpec things" do # "it" comes from refinements
'foo'.should == 'foo' # "should" comes from refinements
end
end

In this case, the parser could still treat all calls following "using" as needing refinements. This is less ideal, because it force us to treat all "using" calls everywhere in the system as potentially triggering refined call sites. It also makes it impossible to detect if "using" is ever aliased or sent. If such features are expected, we'd be back to treating all calls as refined all the time.

Honestly, I really believe that allowing refinements to affect scopes that have no "using" anywhere in sight is going to be really confusing, and it has the added pain of making efficient call site caching hard or impossible.

#117 - 11/05/2012 03:07 PM - shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

> I agree with Charles here. Avoiding global cache invalidation is priority one.
> At the same time, I want a way to enable refinement to given block.
>
> Matz.

At first, I expected using is called only at an early stage of a program execution.
However, this expectation doesn't hold true if module_eval is affected by refinements.

I guess invalidation of the global method cache can be avoided if a refinement table (nd_refinements) and global cache entries have a "refinement version". It's enough to increment the refinement version when the refinement table is changed.

#118 - 11/05/2012 03:17 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> I currently don't see any way to support enabling refinements in blocks without a global, unavoidable impact to all call sites.
>
> If refinements are applied lexically, then a block suddenly having refinements active does not make sense, since the code in the block is not lexically surrounded by a scope with active refinements.
>
> If refinements are applied based on the cref module, then a block still can't see them because cref for the block is determined lexically. This was explicitly done to allow constant lookup to always proceed based on lexical scopes.

In Ruby 1.9, a new cref node with the special flag (NODE_FL_CREF_PUSHED_BY_EVAL) is pushed by module_eval. The cref node is used to determine where a method is defined by def, and is skipped for a constant lookup.

Is it hard to implement JRuby's cref equivalent (StaticScope?) in the same way?

#119 - 11/05/2012 04:23 PM - ko1 (Koichi Sasada)

(2012/11/04 5:57), SASADA Koichi wrote:

> I have another implementation idea, using "prepend".

Nobu taught me that this approach is not work.

--
// SASADA Koichi at atdot dot net

#120 - 11/07/2012 09:23 AM - Anonymous

On Mon, Nov 05, 2012 at 02:17:49PM +0900, headius (Charles Nutter) wrote:

> Issue #4085 has been updated by headius (Charles Nutter).
>
> I currently don't see any way to support enabling refinements in blocks without a global, unavoidable impact to all call sites.
>
> If refinements are applied lexically, then a block suddenly having refinements active does not make sense, since the code in the block is not lexically surrounded by a scope with active refinements.
>
> If refinements are applied based on the cref module, then a block still can't see them because cref for the block is determined lexically. This was explicitly done to allow constant lookup to always proceed based on lexical scopes.
>
> Formally, the clearest way for refinements to work is for you to explicitly opt into them everywhere you want to use them. A couple thoughts about different forms:
>
> 1. using as a scope-opening keyword like "class" or "module":
>
> using RSpec
> describe "Some Spec" do
> it "does some RSpec things" do # "it" comes from refinements
> 'foo'.should == 'foo' # "should" comes from refinements
> end
> end
> end
>
> This is clear to the user and the parser that all calls downstream from the "using" must consider refinements. They would see the refinements based on cref (or something similar) because of the explicit nesting of scopes.
>
> 1. "using" as a pseudo-keyword, affecting lexical scopes and code following "using"
>
> using RSpec
>
> describe "Some Spec" do
> it "does some RSpec things" do # "it" comes from refinements
> 'foo'.should == 'foo' # "should" comes from refinements
> end

> end
>
> In this case, the parser could still treat all calls following "using" as needing refinements. This is less ideal, because it force us to treat all "using" calls everywhere in the system as potentially triggering refined call sites. It also makes it impossible to detect if "using" is ever aliased or sent. If such features are expected, we'd be back to treating all calls as refined all the time.
>
> Honestly, I really believe that allowing refinements to affect scopes that have no "using" anywhere in sight is going to be really confusing, and it has the added pain of making efficient call site caching hard or impossible.

If I'm understanding you correctly, the thing I don't like about
explicitly requiring people to add "using" means that we can't remove
the monkey patches from Rails without breaking everything.

For example:

class MyModel < ActiveRecord::Base
def foo
# constantize comes from Rails monkey patches on String
"somestring".constantize
end
end

Ideally, in the Rails source code, I would just add "using StringExt" to
the ActiveRecord::Base class, and the existing user code would Just
Work. If existing Rails users are required to add the "using" word to
all of their code, then there is no way we (the rails team) can remove
the monkey patches and remain backwards compatible.

Being able to remove monkey patches and remain backwards compatible is
the number one most important thing I want from refinements.

--
Aaron Patterson
http://tenderlovemaking.com/

#121 - 11/07/2012 11:19 AM - shugo (Shugo Maeda)

- Assignee changed from matz (Yukihiro Matsumoto) to shugo (Shugo Maeda)

shugo (Shugo Maeda) wrote:

> At first, I expected using is called only at an early stage of a program execution.
> However, this expectation doesn't hold true if module_eval is affected by refinements.
>
> I guess invalidation of the global method cache can be avoided if a refinement table (nd_refinements) and global cache entries have a "refinement version". It's enough to increment the refinement version when the refinement table is changed.

It seems that singleton method definitions and Kernel#extend also invalidate the entire global method cache.
Is it intended?

The current implementation of module_eval doesn't invalidate the global method cache if no refinement is used in the receiver module. Do you want module_eval not to invalidate the global method cache even if refinements are used, Matz?

I wonder which of singleton method definitions and module_eval with refinements are often used in real-world applications.

#122 - 11/07/2012 10:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-11-2012 22:10, Aaron Patterson escreveu:

> On Mon, Nov 05, 2012 at 02:17:49PM +0900, headius (Charles Nutter) wrote:
>
> > Issue #4085 has been updated by headius (Charles Nutter).
> >
> > I currently don't see any way to support enabling refinements in blocks without a global, unavoidable impact to all call sites.
> >
> > If refinements are applied lexically, then a block suddenly having refinements active does not make sense, since the code in the block is not lexically surrounded by a scope with active refinements.
> >
> > If refinements are applied based on the cref module, then a block still can't see them because cref for the block is determined lexically. This was explicitly done to allow constant lookup to always proceed based on lexical scopes.
> >
> > Formally, the clearest way for refinements to work is for you to explicitly opt into them everywhere you want to use them. A couple thoughts about different forms:

1. using as a scope-opening keyword like "class" or "module":

```
using RSpec
describe "Some Spec" do
it "does some RSpec things" do # "it" comes from refinements
'foo'.should == 'foo' # "should" comes from refinements
end
end
end
```

This is clear to the user and the parser that all calls downstream from the "using" must consider refinements. They would see the refinements based on cref (or something similar) because of the explicit nesting of scopes.

1. "using" as a pseudo-keyword, affecting lexical scopes and code following "using"

```
using RSpec

describe "Some Spec" do
it "does some RSpec things" do # "it" comes from refinements
'foo'.should == 'foo' # "should" comes from refinements
end
end
```

In this case, the parser could still treat all calls following "using" as needing refinements. This is less ideal, because it force us to treat all "using" calls everywhere in the system as potentially triggering refined call sites. It also makes it impossible to detect if "using" is ever aliased or sent. If such features are expected, we'd be back to treating all calls as refined all the time.

Honestly, I really believe that allowing refinements to affect scopes that have no "using" anywhere in sight is going to be really confusing, and it has the added pain of making efficient call site caching hard or impossible.
If I'm understanding you correctly, the thing I don't like about
explicitly requiring people to add "using" means that we can't remove
the monkey patches from Rails without breaking everything.

For example:

```
class MyModel<  ActiveRecord::Base
def foo
# constantize comes from Rails monkey patches on String
"somestring".constantize
end
end
```

Ideally, in the Rails source code, I would just add "using StringExt" to
the ActiveRecord::Base class, and the existing user code would Just
Work. If existing Rails users are required to add the "using" word to
all of their code, then there is no way we (the rails team) can remove
the monkey patches and remain backwards compatible.

Being able to remove monkey patches and remain backwards compatible is
the number one most important thing I want from refinements.

I agree with Charles here. I'd love to see Rails moving to a more
explicit approach instead of monkey patching core classes.

It is hard for someone new to Rails to understand what is the Ruby
behavior and what has been changed by ActiveSupport. Once you force
users to include the "using" whenever they're using some AS extension it
would become clear to understand where some methods came from.

I think Rails should change its mindset with regards to all "magic" it
has been providing from the beginning and become closer to Ruby.

It already started doing so by deprecating dynamic finders in AR, which
was a good start. I'd love to see more "magic" to disappear from Rails
as code is easier to understand once it is more explicit about where all
magic is coming from. Rails 4 could be a great opportunity for a new
paradigm like this. So, I'd also suggest you to take the chance to
change the behavior of constantize to be the same as const_get as you
have requested for feedback on Rails-core mailing list.

We should work to reduce the gap between Rails and Ruby and try to make
Rails just a web framework, keeping Ruby classes unpatched.

Cheers,
Rodrigo.

#123 - 11/13/2012 10:57 AM - headius (Charles Nutter)

Well, I have some bad news.

I have spent some time trying to find a reasonable way to implement refinements in JRuby, and without reducing the feature set it's simply not possible to do without global (and sometimes terrible) impact to general application performance.

At this point I'm largely of the opinion that pure-lexical, syntactical refinements are the way to go. And there are several reasons for it:

- Dynamic refinements have far-reaching impact, both peformance-wise and "understandability" of the code. It's both hard (or impossible) to optimize and hard (or impossible) to reason about.
- Syntactic/lexical refinements are nearly self-explanatory in code. Currently, when you look at a piece of code with method calls (and constant lookup, as well), you can reason about it in terms of the enclosing classes/modules and basically know what it will call. Monkeypatching can obscure the eventual method called, but every piece of code everywhere will still call the same method. With dynamic refinements, it's impossible to look at a piece of code and the classes involved and know what method will be called, because it can change arbitrarily at runtime.
- The current refinements are poorly specified and have only been explored in a very limited way. This is of *great* concern to me. There are a huge number of potential edge cases, ranging from abuse of the module_eval feature to reopening existing refinements (which essentially just makes monkeypatching harder to discover), and implementation-wise there has not been enough exploration of how it will affect current and future performance of Ruby. We need more time to define what refinements should do in terms of what users want and what's reasonable from a language-design perspective.

I would support the idea of pure lexical/syntactic refinements of the following form, but I don't see a way to support the features of the current refinements implementation reasonably on any VM.

module Foo
class Quux
using Bar, Baz
def blah
'abc'.efg # refined
end
end

```
def blah2
  'asdf'.qwer # unrefined
end
```

end
end

#124 - 11/13/2012 11:04 AM - headius (Charles Nutter)

Replies to recent comments:

Aaron: I totally understand the use case, and I support it. Unfortunately I don't feel like the use case and the current feature set have been aligned properly without major impact to unrefined code. There's a certain realism we need here...the feature itself may be useful, but drastically altering the way Ruby does method lookup is a very blunt way to add it. Refinements as they are currently implemented in trunk add complexity well beyond simply localizing monkey-patches, and I would argue that that complexity is a bigger risk than monkey patches ever were. This is of course ignoring the fact that it may be impossible to implement refinements efficiently given the current feature-set.

Rodrigo: You agree with me, so there's not a lot to say :-) I too would like to see Rails move away from magically patching classes. For example, the following code is really no worse then the monkey-patched version:

class Foo
include CamelizeString
def bar(str)
camelize(str)
end
end

versus

class Foo
def bar(str)
str.camelize
end
end

We're talking about a difference of a single character here, but the non-monkeypatched code is actually clearer (you know to look in CamelizeString for the camelize logic) and less invasive (only code that calls CamelizeString#camelize will hit that code). This may be a bit of a paradigm shift for Rails, but I think that's exactly what's needed to make monkey-patching GO AWAY rather than trying to find cute ways to localize monkeypatching.

#125 - 11/13/2012 12:11 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> Well, I have some bad news.
>
> I have spent some time trying to find a reasonable way to implement refinements in JRuby, and without reducing the feature set it's simply not possible to do without global (and sometimes terrible) impact to general application performance.

ko1 came up with a new idea to implement refinements without impact to applications which don't use refinements.
The basic idea is:

- When a method is defined in a refinement (a module given as self in a block of Module#refine), add a method entry with the special type such as VM_METHOD_TYPE_REFINED to the class to be refined. If the original class has a method with the same name, the original method is linked to the new method entry with VM_METHOD_TYPE_REFINED.
- If and only if a method entry with VM_METHOD_TYPE_REFINED is found in a method invocation, search methods of refinements. However, even if a method is found in refinements, don't store the method of refinements in an inline method cache. Instead, store the method entry with VM_METHOD_TYPE_REFINED. (So it may be better to have a cache dedicated to refinements) If no method is found in refinements, use the original method entry linked to the method entry with VM_METHOD_TYPE_REFINED.

What do you think of this idea?

#126 - 11/13/2012 04:42 PM - jballanc (Joshua Ballanco)

Perhaps I am missing something, but for the case that Aaron points out, it seems to me that refinements are an overly complicated solution. I think a much better solution would be to make literal construction use current scope when assigning class. For example:

```
class Foo
String = ::String.dup
class String
def upcase
self.downcase
end
end

def say
String.new("Hello!").upcase #=> Imagine if "Hello!" and String.new("Hello!") were equivalent
end
end

class Bar < Foo
def shout
String.new("HELLO!").upcase
end
end

"test".upcase #=> "TEST" (Foo's monkey patching doesn't leak)

Foo.new.say #=> hello!

Bar.new.shout #=> hello! (monkey patch is inherited)
Bar.new.shout.class #=> Foo::String (class name tells us exactly where to look for patched method definitions)
```

The only thing that would need to change to make this work would be for litteral construction to inspect current scope (similar to simply typing an unqualified "String.new"). I suppose this would be a backwards incompatible change, but it seems to me like an easier solution to implement and work with...

#127 - 11/13/2012 07:29 PM - jballanc (Joshua Ballanco)

Ok, I think I finally figured out what it is about refinements that makes me so uncomfortable... Refinements violate PLOS in a very bad way, I think, because the behavior of code no longer depends only on the objects and statements in the code, but also where that code is located. For example, consider the following:

```
module RefString
refine String do
def camelize
"I'm a camel!"
end
end
end

class ARBase
using RefString
```

```
end

class DoIt < ARBase
def handle(a_string)
puts a_string.camelize
# ... other stuff
end
end
```

Now, consider the very common case where we would refactor this code by extracting a portion of the method into a helper class:

```
class DoIt
def handle(a_string)
Helper.new.help_me(a_string)
# ... other stuff
end
end

class Helper
def help_me(a_string)
puts a_string.camelize
end
end
```

This, of course, breaks...and I worry that this will have a chilling effect on the willingness of people to refactor their code (something we all don't do enough of as is). I think any benefit that library authors gain in not monkey-patching core classes would be outweighed by the reduced ability to do simple refactorings.

#128 - 11/13/2012 08:36 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Charles, although I agree that the example you gave reads just as good as the monkey patched version, sometimes I can't find a better approach to some DSL created by monkey patches. This is why I'd like to have something like refinements, but with a much more strict scope. Consider the usage below from ActiveSupport, which is included by Rails:

expire_at = 2.days.from_now

This is very useful and readable but it requires monkey patching Numeric. But it is not clear by looking at this code where "days" was defined. So, I'd prefer to have something more explicit like:

```
using TimeManipulation do
expire_at = 2.days.from_now
end
```

Groovy implements something like this:

http://groovy.codehaus.org/api/groovy/time/TimeCategory.html

#129 - 11/14/2012 01:35 AM - headius (Charles Nutter)

shugo: I believe I understand the implementation. There are a lot of open questions for it, however:

- If the refined class is reopened and the method redefined, does the VM_METHOD_TYPE_REFINED flag get lost?
- How are refinements looked up? In JRuby, we need to know ahead of time if a method body will have refinements active, or we'll have to have them available all the time...with a perf cost. Moving the trigger to a flag on the target method means we can't do that.
- If a refinement is not present the first time, do we never check again?
- If we do check again, doesn't it mean all call sites would need invalidation for potential refinements on every call?

This implementation doesn't really save us anything because it still requires that all call sites check for refined methods, and we can't tell until call time that they'll need to look refinements up. Ultimately, we have to cripple all code just the same as we did before.

Of course it also doesn't address the fact usability, debuggability issues I mentioned either.

#130 - 11/14/2012 01:43 AM - headius (Charles Nutter)

jballanc: That is exactly the sort of problem I'm worried about. It is *impossible* to look at the DoIt class in your example and know what methods it will call. Even with monkey-patching, you would have the consistency of all callers seeing the same code. Not so with refinements; the location of the code can now change what methods are called. Moving code can change what methods are called. Changing your class hierarchy can change what methods are called. Passing a block of code to a new method can change what methods are called.

I feel more and more like refinements will make monkey-patching *more* confusing and patched code *more* difficult to understand.

rosenfeld: Groovy does indeed have categories (their equivalent of refinements, but thread-local and down-stack), and they have been a real pain point for them improving Groovy. Because they need to be checked all the time, they impact call performance. They've found various tricks to improve that, but they mostly have been trying to move away from them. And I know this from talking to the implementers themselves...they regret ever adding categories for exactly the reasons I have enumerated: they make optimization harder, they make code more difficult to follow, they don't solve as many problems as they create.

#131 - 11/14/2012 02:21 AM - headius (Charles Nutter)

shugo: I may have a possible compromise that fixes some of the technical issues.

Currently, refinements have to be looked up via cref, basically (there's oddities for module_eval case, but it's similar to cref). If instead refinements are located *solely* based on the caller object, we can implement refinements using ko1's method flag trick.

The logic would work like this:

- Any method that gets refined gets flagged, as you described earlier.
- A call site that encounters a refined method at lookup time will do the search for refinements by looking at the calling object.
- Lookup will check the object's class and superclasses searching for the refined version of the method.
- If a refined method is found in the caller's hierarchy, it is cached at the call site, but instead of guarding only on the target class it also guards based on the calling class. Modifications to either invalidate the site.
- If a refined method is not found in the caller's hierarchy, caching proceeds as normal.

The edges here are whether refinements added to a caller's hierarchy later should get picked up. If that was a requirement, it would require all call sites cache based on caller as well, regardless of whether they see refinements the first time or not. The alternative would be like ko1 suggested, not caching refinements at the call site at all and only caching the target method as a trigger to look in a second cache that's invalidated globally when modifications come in.

If refinements added after the first call should not be seen, then it's more like the current "temporal" application of refinements.

This implementation would add no overhead to unrefined call logic (other than the initial check) because it's similar to visibility checking. In order to check visibility, we already have to pass caller to the call site anyway.

I'll note again that this does not make any of the added complexity of refinements (from a user/programmer perspective) go away...it just might make it easier to implement without impacting unrefined calls. It also doesn't address the concerns about future modifications to the refined class and whether they can overwrite previously-refined methods.

#132 - 11/14/2012 05:15 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Charles, I'm not sure if I completely understand your concerns, but I think that from a code readability point of view refinements should be local to the current block or it would become just as difficult to read code as the current monkey-patch approach. See this example:

def a
using TimeConversion do
2.days.ago # OK
b # see below
end
end

def b
2.days.ago # should raise a NoMethodError exception in my opinion
end

If the behavior for refinements were like described above, would it still impact on performance optimizations?

I didn't test something like this in Groovy, but I'm assuming "b" wouldn't raise any exceptions in a situation like above, right? Maybe that is why it is hard to improve performance there...

#133 - 11/14/2012 05:18 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Just tested it to confirm:

a = {1.hour.ago}; use(groovy.time.TimeCategory){a()}

If "a()" throwed an exception in Groovy for the example above would it be easier for them to optimize the interpreter?

#134 - 11/14/2012 05:23 AM - headius (Charles Nutter)

Talking more with folks online about refinements, there's a lot of confusion about what they solve.

So, in one sense, refinements are to localize monkey-patching. But they don't actually localize it much better since they can apply at a distance to blocks (module_eval feature), and classes down-hierarchy.

Previously, all code determined what methods to call based solely on the target object's class hierarchy. Even with monkeypatches in place, we still have to look solely at the target class to determine what's being called.

With refinements, every piece of code everywhere in the system could potentially see refinements active whether there's a "using" clause near them or not. Blocks could be forced to call different methods at any time, normal code could see a superclass add a refinement and change all future calls. Refinements may prevent monkeypatches from affecting the entire runtime, but don't make it any easier to determine what methods will actually be called.

They also don't solve the monkeypatching problem in any way. Monkeypatches have been used for a few reasons:

- Adding methods to existing types, for DSL or fluent API purposes.

Refinements would limit the visibility of those methods, somewhat, but you can't tell without digging around both the target class hierarchy and the calling class hierarchy what methods will really be called.

- Replacing existing methods with "better" versions

Refinements would again limit the visibility of those changes, but ultimately result in some code calling one method and some code calling another, with no easy way to determine the code that will be called ahead of time.

It may be possible to address the technical issues of optimizing call sites with and without refinements, but I really don't feel like refinements are solving as many problems as they're going to create. I lament a future where I can't look at a piece of code and determine the methods it's calling solely based on the types it is calling against. It's going to be harder -- not easier -- to reason about code with refinements in play.

#135 - 11/14/2012 05:38 AM - headius (Charles Nutter)

rosenfeld: Yes, I am arguing that same case. I believe refinements should only be active for code that appears within a refined context. My example from earlier:

```
class Foo < SomeParent
def bar(str)
str.upcase # unrefined
end

using StringRefinement
def baz(str)
str.camlize # refined
end
end
end
```

There are implementation reasons why this is simpler, but the more important reasons are readability, understandability of the code. You know exactly whose methods will be called in both cases -- String's in the bar() body and StringRefinement's or String's in the baz() body -- and there's no question whether refinements are active for a given call. Compare that to the following code:

```
class Foo < SomeParent
def bar(str)
str.upcase
end

def baz(str)
str.camelize
end
end
```

What methods are being called? Where are they coming from? You can't know, since you need more information than the type of object that str is. You need to know whether Foo has previously had refinements applied, whether SomeParent previously had refinements applied, whether its parents previously had refinements applied...you need to know what those refinements are and whether they affect String methods...and you need to know whether any of the methods you are calling have been refined.

EVERY PIECE OF CODE in a given system now forces users to understand BOTH the target class being called AND the hierarchy of code surrounding the call. That's not simpler, it's more complicated...and it affects the readability of ALL CODE.

And then there's this:

```
class Foo < SomeParent
def baz(str)
ary.map {|name| str.camelize + name}
end
end
```

In this case, you have to check even more places for refinements to know what methods will be called:

- Foo may have been previously refined. You must look for all reopenings of Foo to know what will be called.
- SomeParent or its parents may have been previously refined. You must look for all reopenings of SomeParent and its parents.
- The map method may force refinements on the block. you must look for all implementations of map() that might be called here to see if they

> force refinements into the block.

This is supposed to be simpler?

#136 - 11/14/2012 07:19 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

It reminds me C++. So powerful that you can have a hard time trying to understand what is happening in a given piece of code since you have so many operators and each of them can be overloaded. Very powerful but once you're relying on some third-party library using those powerful features instead of using them yourself you can find yourself in trouble trying to understand what is going on with your code.

#137 - 11/14/2012 03:43 PM - brainopia (Ravil Bayramgalin)

> Ideally, in the Rails source code, I would just add "using StringExt" to
> the ActiveRecord::Base class, and the existing user code would Just
> Work. If existing Rails users are required to add the "using" word to
> all of their code, then there is no way we (the rails team) can remove
> the monkey patches and remain backwards compatible.

Aaron, if refinements were not leaking to inherited classes (as Yehuda nicely described why) then you still would have the ability to manually propagate them if needed, eg:

class ActiveRecord::Base
using ActiveRecord::Refinements

def self.inherited(klass)
klass.module_eval { using ActiveRecord::Refinements }
end
end

#138 - 11/15/2012 03:45 AM - Anonymous

On Wed, Nov 14, 2012 at 05:38:12AM +0900, headius (Charles Nutter) wrote:

[snip]

> And then there's this:
>
> class Foo < SomeParent
> def baz(str)
> ary.map {|name| str.camelize + name}
> end
> end
>
> In this case, you have to check even more places for refinements to know what methods will be called:
>
> - Foo may have been previously refined. You must look for all reopenings of Foo to know what will be called.
> - SomeParent or its parents may have been previously refined. You must look for all reopenings of SomeParent and its parents.
> - The map method may force refinements on the block. you must look for all implementations of map() that might be called here to see if they force refinements into the block.
>
> This is supposed to be simpler?

I have to agree. It seems like this would be *much* more difficult to
debug than if camelize is just monkey patched on to String.

This code:

class Foo < SomeParent
def baz(str)
cached = str.camelize
ary.map {|name| cached + name}
end
end

Could have a completely different meaning than this code:

class Foo < SomeParent
def baz(str)
ary.map {|name| str.camelize + name}
end
end

That seems extremely bad.

--
Aaron Patterson
http://tenderlovemaking.com/

#139 - 11/15/2012 08:29 AM - trans (Thomas Sawyer)

Perhaps refinements should be scoped per-gem, rather than any arbitrary
"using" delimitation. Seems to me, that is generally the level at which we
care about them.

Would that simplify implementation and comprehensibility of usage to
something more manageable?

On Wed, Nov 14, 2012 at 1:35 PM, Aaron Patterson
tenderlove@ruby-lang.orgwrote:

> On Wed, Nov 14, 2012 at 05:38:12AM +0900, headius (Charles Nutter) wrote:
>
> [snip]
>
>> And then there's this:
>>
>> class Foo < SomeParent
>> def baz(str)
>> ary.map {|name| str.camelize + name}
>> end
>> end
>>
>> In this case, you have to check even more places for refinements to know
>> what methods will be called:
>>
>> - Foo may have been previously refined. You must look for all reopenings of Foo to know what will be called.
>> - SomeParent or its parents may have been previously refined. You must look for all reopenings of SomeParent and its parents.
>> - The map method may force refinements on the block. you must look for all implementations of map() that might be called here to see if they force refinements into the block.
>>
>> This is supposed to be simpler?
>
> I have to agree. It seems like this would be *much* more difficult to
> debug than if camelize is just monkey patched on to String.
>
> This code:
>
> class Foo < SomeParent
> def baz(str)
> cached = str.camelize
> ary.map {|name| cached + name}
> end
> end
>
> Could have a completely different meaning than this code:
>
> class Foo < SomeParent
> def baz(str)
> ary.map {|name| str.camelize + name}
> end
> end
>
> That seems extremely bad.
>
> --
> Aaron Patterson
> http://tenderlovemaking.com/

--
Sorry, says the barman, we don't serve neutrinos. A neutrino walks into a
bar.

Trans transfire@gmail.com
7r4n5.com http://7r4n5.com

#140 - 11/17/2012 12:53 PM - The8472 (Aaron G)

> So, in one sense, refinements are to localize monkey-patching. But they don't actually localize it much better since they can apply at a distance to blocks (module_eval feature), and classes down-hierarchy.
> Previously, all code determined what methods to call based solely on the target object's class hierarchy. Even with monkeypatches in place, we still have to look solely at the target class to determine what's being called.
> With refinements, every piece of code everywhere in the system could potentially see refinements active whether there's a "using" clause near them or not. Blocks could be forced to call different methods at any time, normal code could see a superclass add a refinement and change all future calls. Refinements may prevent monkeypatches from affecting the entire runtime, but don't make it any easier to determine what methods will actually be called.

I think I might have a solution to this:

1. refinements should only go through the local module hierarchy, not the class hierarchy because all contributors to a module namespace should be familiar with the conventions and refinements used within that module.

If for example ActiveRecord wants to add .constantize their Strings and
use that feature throughout their project it should be fine, as they
should be putting everything under the ::ActiveRecord module anyway.

If someone has his own rails application he can use the refinement again
in his own application module or - if he isn't using any module - apply
it to all of his classes.

1. instance_eval/module_eval/class_eval should NOT apply the refinements of the target. Instead they should use the same refinements as the context they were defined in.

To still allow for fancy DSLs there should be a way to explicitly rebind
the context of the Proc to a different refinement context. Basically
bind its lookup to a different module.

A Proc passed to the DSL would first have to be re-bound to the DSL's
own module and then eval'd. As long as nobody else rebinds the Proc
again this shouldn't invalidate any cache as the Proc was never called
in its old context before. What is even better is that someone else
could extend the DSL under a different module but the DSL's extensions
would still get applied to the block, as expected by the caller.

1. there should also be an way to remove refinements from a module and all its submodules. This makes it possible to apply a refinement to ::Object and then prune it out of some foreign namespaces where it turns out to cause trouble.

Consider it "optimistic refining".

1 and 2 mean that every Callsite and Proc can only have one refinement
context at any given time.
Rule 3 makes monkeypatching safer as we can still apply it globally as
we already do and simply undo it in places where the monkey bites us.

While this may seem less flexible you also have to consider that these
single-scope refinements can be used together with Module.prepend

This allows you to move an aspect of code to a different module inside
your application, apply refinements only to that module (and its
submodules), write your refined code inside that module and then prepend
that module with the refined behavior to the target class which you do
not want to "pollute" with the refinements.

> They also don't solve the monkeypatching problem in any way. Monkeypatches have been used for a few reasons:
>
> - Adding methods to existing types, for DSL or fluent API purposes.
>
> Refinements would limit the visibility of those methods, somewhat, but you can't tell without digging around both the target class hierarchy and the calling class hierarchy what methods will really be called.
>
> - Replacing existing methods with "better" versions
>
> Refinements would again limit the visibility of those changes, but ultimately result in some code calling one method and some code calling another, with no easy way to determine the code that will be called ahead of time.
>
> It may be possible to address the technical issues of optimizing call sites with and without refinements, but I really don't feel like refinements are solving as many problems as they're going to create. I lament a future where I can't look at a piece of code and determine the methods it's calling solely based on the types it is calling against. It's going to be harder -- not easier -- to reason about code with refinements in play.

#141 - 11/20/2012 11:45 AM - shugo (Shugo Maeda)

- Assignee changed from shugo (Shugo Maeda) to matz (Yukihiro Matsumoto)

Thanks for your feedback, Charles and others.
I understand your worries.

The feature set of Ruby 2.0 has already been frozen, so it's impossible to introduce a completely different feature in Ruby 2.0. So we have only the following options:

1. introduce the whole features of Refinements currently implemented
2. introduce some of the features of Refinements (= drop some features)
3. remove all features of Refinements

I think optional features of Refinements are as follows:

A. refinement inheritance in class hierarchies
B. refinement activation for reopened module definitions
C. refinement activation for the string version of module_eval/instance_eval
D. refinement activation for the block version of module_eval/instance_eval

I've asked Matz to decide whether Refinements should be included in Ruby 2.0, and if so, which of these features should be included.

My own take is as follows:

- I'm not sure A is good or not. It's useful in some cases, but it may be confusing because include doesn't inherit refinements, but class inheritance does. So it's OK to remove A from Ruby 2.0.
- I want C and D for internal DSLs, but D might be difficult to implement in VMs other than CRuby. So it's OK to remove D from Ruby 2.0. FYI, I'm implementing it without performance overhead when refinements are not used. http://shugo.net/tmp/refinement_fix_1119.diff In this implementation, refined methods are stored in neither an inline method cache nor the global method cache, so there's no need to invalidate cache for module_eval. I hope D will be introduced in the future.
- From the perspective of consistency, C and D depend on B. So if C or D is included in Ruby 2.0, B should also be included.

And, I explain some things to clarify my intention.

I have used the word "lexical" to describe Refinements, but by the word I've meant just that Refinements doesn't support local rebinding. For example, in the following code, FooExt doesn't affect Bar#call_foo even if it's called from Baz, which is a module using FooExt.

```
class Foo
end
module FooExt
refine Foo do
def foo
puts "foo"
end
end
end
class Bar
def call_foo(f)
f.foo
end
end
module Baz
using FooExt
f = Foo.new
f.foo              # => foo
Bar.new.call_foo(f)  # => NoMethodError
end
```

I think it's the most important feature of Refinements. Without it, it's hard to avoid conflicts among multiple refinements.

Some people seem to suspect that code using refinements is difficult to debug, but reflection APIs may be useful to debug such code.

```
module M
refine Fixnum do
def foo; puts "foo" end
end
end
using M
p 123.method(:foo).owner #=> #refinement:Fixnum@M
```

I admit that Refinements are complex, but it's because issues to address by Refinements are themselves complex. And, I think Refinements should not be over-used. Application programmers should not use Refinements. Refinements are for library/framework programmers. Besides, even if you're a library or framework programmer, consider other features such as subclassing before Refinements.

#142 - 11/20/2012 01:40 PM - trans (Thomas Sawyer)

=begin
So...

class Foo
end
module FooExt
refine Foo do
def foo
puts "foo"
end
end
end
module Kernel
def safe_call(f, m)
if f.respond_to? m
return f.send(:m)
end
nil
end
end
class Object
def safe_send(m)
if respond_to? m
return f.send(:m)
end
nil
end
end
module Baz
using FooExt
f = Foo.new
f.respond_to? :foo # => true
safe_call(f, :foo) # => nil
f.safe_send(:foo)   # => ?
end

And how much worse if:

class Object
def safe_send(m)
safe_call(self, m)
end
end
=end

#143 - 11/20/2012 02:48 PM - shugo (Shugo Maeda)

shugo (Shugo Maeda) wrote:

> Application programmers should not use Refinements.

I meant that application programmers should not use Module#refine.
It's OK to use Kernel#using and Module#using.

#144 - 11/20/2012 02:52 PM - shugo (Shugo Maeda)

trans (Thomas Sawyer) wrote:

> ```
> f.safe_send(:foo)   # => ?
> ```

This should return nil. Otherwise, refinements can break code which doesn't expect the refined behavior.

> And how much worse if:
>
> class Object
> def safe_send(m)
> safe_call(self, m)
> end
> end

I don't understand why it's worse.

#145 - 11/20/2012 08:24 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

shugo (Shugo Maeda) wrote:

> Some people seem to suspect that code using refinements is difficult to debug, but reflection APIs may be useful to debug such code.

I don't think the big problem is debugging, but readability. This is my own definition of readability: the longer it takes for one to understand some code, the less readable it is. In that sense, refinements in libraries seem even less readable than refinements in application code.

Consider someone new to Ruby reading a Rails code like "some_string.camelize". He could look for camelize in String class RDoc and he wouldn't find it there. He would be forced to use the reflection API to understand where is that method coming from. This reduces readability as the developer now needs an extra step (time) to figure out where that method is coming from. Looking at the API is quick. Debugging or using the reflection API is not that quick. That is the same problem I have with Monkey Patches. They reduce readability. It is even worse when it overrides some built-in method and change its behavior.

That is the same reason I don't add extensions of my own to core classes (like aliasing each_with_object to a shorter name, etc). It makes it harder for others in the team to read the application code.

I'm pretty sure that once refinements are included in final 2.0.0 people will start using it just because they want to use the new fancy feature and not because they really feel its need. This is what probably happened when DHH first introduced dynamic finders in ActiveRecord in my opinion. I believe he found it fantastic the method_missing feature and decided to use it just because he could. There was no real need for that. And people keep saying that it is ok to use monkey patches and method_missing at will and that Ruby even encourages that practice or otherwise those features wouldn't exist. People build some culture over what is Ruby best practices and what is not. I remember that someone from the Rails core team judging my patch some years ago stating that I shouldn't use "is_a?" because the Ruby way is to use duck-typing. Even so I really wanted to test for the specific class instead of just asking if the object responds to some method. It was much more readable to use "is_a?" in that context.

I'm afraid that once refinements are possible in Ruby they will be immediately abused just because they are fancy and it will take years for libraries to start to avoid them because they are no longer fancy (like what happened to ActiveRecord removing the dynamic finders just now, years after it has been introduced to AR).

I know this is hard to balance. People moved from other languages to Ruby because Ruby was a more powerful language. But that is not the single reason. It was also more readable. But a feature like this one has its tradeoffs. It makes Ruby more powerful, by reducing conflicts that might be created by conflicting versions of monkey patches. But at the cost of both performance (which I think is the least problem here) and readability.

The worst part for me is that even if I opt out for using refinements myself, I'll still have to live with it and create tons of checks when debugging code and trying to understand others' (libraries/frameworks/applications) code. I realize this issue is not introduced by this feature as monkey patches have the same effect. But the problem this feature introduces is that it becomes even harder to read code once refinements are possible.

For instance, with monkey patches, if I just make sure I load/require all files in the same order, I could write a code using the reflection API to understand where a method is coming from. But now this won't suffice anymore because the same method could be defined elsewhere when inside another context. If that other context is not easily triggered/reproducible than it gets even hard to try to understand why that special condition is not working under production, for instance. Imagine yourself trying to fix a bug caused by some race-condition that you can't always replicate. Usually some code inspection is all that you get left to work on. But once refinements are possible, inspecting some code becomes so much more difficult to perform :(

Then, the real question is: what is most important? Powerful or readability/performance? We can't get both with this feature.

#146 - 11/20/2012 10:32 PM - shugo (Shugo Maeda)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Then, the real question is: what is most important? Powerful or readability/performance? We can't get both with this feature.

I proposed Refinements for power. Refinements are more powerful than monkey patching in the sense that incompatible class extensions can be mixed in a single program without conflicts.

But it doesn't mean I don't care readability, just mean I prioritize power over readability in this case.

Without module_eval and refinement inheritance in class hierarchies, refinements don't decrease readability so much, because using is used in a file you're reading, except when a module is reopened, in which case you have to pay a price for monkey patching.

In terms of module_eval, expected use cases are internal DSLs. They might look magical, but users don't need to understand the whole stuff in the underlying implementation of a DSL.

I don't see any enough reason to justify refinement inheritance in class hierarchies. That's why I withdrew it in [ruby-core:49631]#141.

Finally, as to performance, it's important that there's no overhead when refinements are not used. I'm working to achieve it, and believe it's possible with ko1's help at least in CRuby.

#147 - 11/20/2012 11:36 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

shugo (Shugo Maeda) wrote:

> ...Without module_eval and refinement inheritance in class hierarchies, refinements don't decrease readability so much, because using is used in a file you're reading, except when a module is reopened, in which case you have to pay a price for monkey patching.

Since this introduces no more harm then monkey patching and since monkey patching isn't going anywhere, I don't object refinements for the implementation strategy you just described if I understand it correctly.

> In terms of module_eval, expected use cases are internal DSLs. They might look magical, but users don't need to understand the whole stuff in the underlying implementation of a DSL.

This is true until something goes weird and you don't know if there is a bug in the DSL implementation and you have to dig on that to understand what is happening. That is the problem with black boxes. Once they are no longer black boxes to you (you need to dig its source-code) it helps when it doesn't use much magic (it gets more readable).

I'd like to state that I'm not against DSLs. I'm all for them. But I wouldn't mind if I had to write my specs like this:

RSpec.describe "Some use case" do |spec|
spec.example "some example" do |dsl|
dsl.expect(something).to happen
end
end

Or even something like:

class << RSpec::Description("Some use case") # RSpec::Description would return an anonymous class, which we're extending below
pending "some example" # pending is defined in RSpec::Description() returned class

end

I just don't think that lots of powerful features would justify some readability loss. I know this is very subjective as lots of developers would find the current DSL for RSpec better than the ones above, but for me I prefer explicit things over implicit ones. Just because it is much easier for me to understand code written this way when bad things happen. Black boxes are painful to work with when things goes wrong in unexpected ways.

#148 - 11/20/2012 11:51 PM - trans (Thomas Sawyer)

=begin

> This should return nil. Otherwise, refinements can break code which doesn't expect the refined behavior.

Ok. Makes sense.

> I don't understand why it's worse.

Only worse in that case if you had not said nil.

Since you did say nil, then what about:

```
class S
  def foo
    "foo"
  end
  def bar
    "bar"
  end
  def foobar
    foo + bar
  end
end

module R
  refine S do
    def bar
      "bar!"
    end
  end
end

class C < S
  using R
  alias :foobar! :foobar
  def foobar?
    foo + bar
  end
  def brainfart1
    c = C.new
    c.foobar! == c.foobar?
  end
  def brainfart2
    c = C.new
    s = S.new
    c.foobar! == s.foobar
  end
  def brainfart3
    c = C.new
    s = S.new
    c.bar == s.bar
  end

end

C.new.brainfart1  #=> ?
C.new.brainfart2  #=> ?
C.new.brainfart3  #=> ?
```

=end

#149 - 11/20/2012 11:53 PM - trans (Thomas Sawyer)

> This should return nil. Otherwise, refinements can break code which
> doesn't expect the refined behavior.

Ok. Makes sense.

> I don't understand why it's worse.

Only worse in that case if you had not said nil.

Since you did say nil, then what about:

```
class S
  def foo
    "foo"
  end
  def bar
    "bar"
  end
  def foobar
    foo + bar
  end
end
```

```
module R
  refine S do
    def bar
      "bar!"
    end
  end
end
```

```
class C < S
  using R
  alias :foobar! :foobar
  def foobar?
    foo + bar
  end
  def brainfart1
    c = C.new
    c.foobar! == c.foobar?
  end
  def brainfart2
    c = C.new
    s = S.new
    c.foobar! == s.foobar
  end
  def brainfart3
    c = C.new
    s = S.new
    c.bar == s.bar
  end

end

C.new.brainfart1  #=> ?
C.new.brainfart2  #=> ?
C.new.brainfart3  #=> ?
```

#150 - 11/21/2012 08:23 AM - The8472 (Aaron G)

On 20.11.2012 03:45, shugo (Shugo Maeda) wrote:

> I think optional features of Refinements are as follows:
>
> A. refinement inheritance in class hierarchies

I generally think that class/module inheritance is the wrong propagation
strategy for refinements. If you see refinements as monkey patches they
are only necessary to get *your own code* working as desired. When you
provide an abstract class/module in a library that application code can
inherit from, then the application itself may not need the refinements
you use to make the internals of your class tick.

In other words, module/class inheritance is about inheriting desired
behavior. Monkey patches may not be desired behavior, they are the
dirty internal mechanics that should be hidden from subclasses.

There is an axis orthogonal to behavior. It's responsibility. This
second axis is generally associated with the module namespaces. E.g. I
expect the rails maintainers to be responsible for the ::ActiveRecord
namespace and be aware of what their own code is doing. Their own
refinements should not pose a problem to them. But they might be for me.

If they "need" a String.camelize in their code then they should be able
to add it without polluting my code. If I consider it useful I can still
include their refinements into my code.

Therefore I think that class inheritance should be removed. And if it
gets replaced in the future then it should be with submodule based
inheritance.

The other issue i have with inheritance is that there is no opt-out.

This is the very same issue we're trying to fix! If some piece of code
monkey-patches Object then the whole application is hit by that
modification. Refinements are supposed to prevent this. But what happens
if i want to use a module that applies refinements? Then I would get hit
by those refinements too.

If we want inheritance then we need some way to opt-out of refinements.
Consider someone applying "using" to Object itself early on during the
application loading process. We would be in the same mess we are in now.
Actually. It would be worse, some methods might see the refinements and
others don't, depending on their definition time.

For now people can use Module.extended/.included if they really want to
add refinement inheritance themselves.

> B. refinement activation for reopened module definitions
> C. refinement activation for the string version of module_eval/instance_eval
> D. refinement activation for the block version of module_eval/instance_eval

I don't feel strongly about those, but if the module_eval performance
really has such a big issue as headius asserts then it might be better
to postpone it until a solution has been found.

Probably the safest approach for now would be to use the source
refinement scope (which is quasi-static) for module_eval by default and
add a way to use the target scope (or an explicit scope) later on as
needed. If there is any performance impact it would restricted to the
target-scoped procs.

I think some clever optimizations should be able to eliminate cases
where procs flow through consistent code paths, i.e. are always
evaluated against the same target refinement scope as usually is the
case with DSLs or class-level configurations.

> I have used the word "lexical" to describe Refinements, but by the word I've meant just that Refinements doesn't support local rebinding. For example, in the following code, FooExt doesn't affect Bar#call_foo even if it's called from Baz, which is a module using FooExt.
>
> class Foo
> end
> module FooExt
> refine Foo do
> def foo
> puts "foo"
> end
> end
> end
> class Bar
> def call_foo(f)
> f.foo
> end
> end
> module Baz
> using FooExt
> f = Foo.new
> f.foo   # => foo
> Bar.new.call_foo(f) # => NoMethodError
> end
>
> I think it's the most important feature of Refinements. Without it, it's hard to avoid conflicts among multiple refinements.

What about cases like

```
module SomeExt
  refine String do
    def bar
    end
  end
end

class Foo
  using SomeExt

  def self.test1
    "".tap(&:bar)
  end

  def self.test2
    "".tap{|f| f.bar}
  end
end
```

String.bar is only visible inside Foo, but in test1 the Proc is created
in .to_proc of Symbol, i.e. on a different stack frame, which shouldn't
be able to see bar due to the scoping. Which leads to counter-intuitive
results.

#151 - 11/21/2012 02:48 PM - shugo (Shugo Maeda)

trans (Thomas Sawyer) wrote:

> Since you did say nil, then what about:
> (snip)
> C.new.brainfart1 #=> ?

false. Because bar in C#foobar? is refined by R, but bar in C#foobar!, which is an alias of S#foobar, is not refined.

```
C.new.brainfart2  #=> ?
```

true. Because both bar in C#foobar! and bar in S#foobar are not refined.

```
C.new.brainfart3  #=> ?
```

true. Because both c.bar and s.bar calls bar refined by R.

Refinements are available in trunk, so could you try it yourself?
If the behavior is unexpected, please report it with a reason why you think it's wrong.

#152 - 11/21/2012 03:15 PM - shugo (Shugo Maeda)

The8472 (Aaron G) wrote:

> Therefore I think that class inheritance should be removed. And if it gets replaced in the future then it should be with submodule based inheritance.

I'll remove it if permission granted by Matz.

> For now people can use Module.extended/.included if they really want to add refinement inheritance themselves.

Currently this wouldn't work because you cannot get the caller context in these hooks.

> > B. refinement activation for reopened module definitions
> > C. refinement activation for the string version of module_eval/instance_eval
> > D. refinement activation for the block version of module_eval/instance_eval
>
> I don't feel strongly about those, but if the module_eval performance really has such a big issue as headius asserts then it might be better to postpone it until a solution has been found.

A solution has been found at least in CRuby.
I'm waiting for ko1's review.

> Probably the safest approach for now would be to use the source refinement scope (which is quasi-static) for module_eval by default and add a way to use the target scope (or an explicit scope) later on as needed. If there is any performance impact it would restricted to the target-scoped procs.

Do you mean that a new option of module_eval should be introduced?
For example,

Foo.module_eval { # use refinements in the current context }
Foo.module_eval(using_refinements: true) { # use refinements in the receiver }

> What about cases like
>
> ```
> module SomeExt
>   refine String do
>     def bar
>     end
>   end
> end
>
> class Foo
>   using SomeExt
>
>   def self.test1
>     "".tap(&:bar)
>   end
>
>   def self.test2
>     "".tap{|f| f.bar}
>   end
> ```

```
end
```

> String.bar is only visible inside Foo, but in test1 the Proc is created
> in .to_proc of Symbol, i.e. on a different stack frame, which shouldn't
> be able to see bar due to the scoping. Which leads to counter-intuitive
> results.

Originally, String#bar was not visible in the Proc created by Symbol#to_proc.
But I've changed it because Matz asked to do. I think the current behavior
is not consistent, but useful.

If Symbol#to_proc were written in Ruby, it would be impossible, but
Symbol#to_proc is written in C. There are some such special methods.
For example, Module.nesting returns the module nesting information in the
caller context. Module#using also affects the caller context.

#153 - 11/21/2012 06:58 PM - duerst (Martin Dürst)

My personal opinion is that at this point in time, there are too many uncertainties surrounding refinements, and so it is too early to include them. I would therefore support "3. remove all features of Refinements", of course putting them back again in trunk once 2.0.0 is forked from trunk, or into a separate branch, so that further work on them can continue.

I also have my doubts about whether they are really that much of a needed. ActiveRecord and friends are often cited. But I wonder what's the problem with ActiveRecord adding #camelize to String. It's a method that's often necessary in metaprogramming, so it might even be added to Ruby core. And str.camelize is much more natural than camelize(str). Good monkey patching is just extending the functionality of base (and other) classes in natural ways. Of course there's bad monkey patching, but there will be bad refinements, too.

So as for other proposals, I think we should work out the use case much more clearly.

(disclaimer: I had a use case, but discussing it with Shugo a bit over a week ago, he told me that this would have been covered by classboxes, but not by refinements)

#154 - 11/21/2012 10:40 PM - trans (Thomas Sawyer)

shugo (Shugo Maeda) Thanks you for answering. I'm still trying to fully understand refinements so it helps to hear some explanation. That the alias is not refined seems maybe a little confusing, since alias effectively copies the method to the current context. Or do I misunderstand the alias spec? But I can see why it is probably best that way regardless.

Refinements scare me a bit. I am not sure what it is exactly, and maybe its just a matter of getting used to them. I guess my biggest question is what happens if Refinements start getting used a lot. Is it possible that this solution to the monkey-patching issue could become worse then the original problem?

#155 - 11/22/2012 04:57 AM - brixen (Brian Shirai)

Please remove Refinements completely from Ruby 2.0 features. They are not well defined, well understood, or well justified.

There are many arguments both for and against. However, the most compelling argument against is that we cannot accurately say what consequences they will have in practice. There is a ton of speculation. That is all.

A feature this far-reaching should not be blindly dumped into the language, especially when the semantics of it are not even fully developed less than three months before the planned release date.

Thanks,
Brian

#156 - 11/22/2012 05:46 AM - headius (Charles Nutter)

I would also vote to remove refinements from 2.0 features, perhaps for reinclusion in 2.1. The various points made about lack of clear specification, lack of time to experiment, lack of clarity on the extent of damage/risk, and lack of time for other implementers to fully implement and explore the feature all point toward this being too much, too late in 2.0's dev cycle...and I believe it is too close to 2.0's release to shove the features in.

That said, I will say I appreciate the reduction in scope. Having refinements only searched via lexical enclosures makes the feature much simpler to implement and much easier to understand in real code. To help the process of fleshing out the feature, I have implemented a large part of the reduced refinements feature for JRuby on the refinements branch: https://github.com/jruby/jruby/tree/refinements

Here are my notes on the current JRuby implementation:

- Only the interpreter is supported for now. Compiler support will require a rework of how we access the scope.
- Much of the determination of whether refinements are active can be static. I set a flag in StaticScope after any call to "using" that flags subsequent calls as refined. This narrows refinement impact to calls following "using".
- I have no strong preference as to whether "refine" additions after a "using" call should be expressed; if they are to be expressed, it means holding a reference to the refinement holder module in the cref; if they are not, it means copying them at the moment of the "using" call.

- Reflective methods (method, instance_method, etc) will have to be special-cased in the refined call site, so that the refinements are looked up. We do not want to have to propagate refinements through to the default implementation of those methods.
- My current implementation caches refinements at the call site based on a global token, incremented whenever a refinement change happens. This allows refined calls to be equivalent performance to unrefined calls in most cases, but it ignores all other invalidation mechanisms (hierarchy changes, etc).
- I am not currently searching modules included into modules for refinements.
- super is not implemented; it is not clear to me how to implement it simply/efficiently in JRuby.

I believe we can implement the lexical-only version with reasonable efficiency. My implementation still follows your basic structure, putting refinements on cref and using anonymous modules to hold them. As mentioned in the notes above, I do not have the special behavior for reflection or super implemented; several of the missing features will be more complicated to complete in JRuby due to the way we optimize code. I will also need to rework the way we handle scoping in JRuby's compiler to efficiently access the cref scope without requiring a full frame, but so far I believe everything is doable.

I still vote to delay adding refinements until after 2.1. It feels very rushed now.

#157 - 11/22/2012 05:47 AM - headius (Charles Nutter)

Sorry, I meant "I still vote to delay adding refinements until after 2.0."

#158 - 11/22/2012 06:07 AM - headius (Charles Nutter)

Here is a trivial benchmark of refined versus unrefined calls in JRuby and MRI trunk. Note that this is only running in JRuby's interpreter, but performance is roughly equivalent between refined and unrefined. I will note again that refined call sites must be globally invalidated right now; this may or may not be a problem for real-world applications.

```
module X
refine Object do
def blah
end
end
end

class String
def blah2
end
end

10.times {
Benchmark.bmbm {|bm|
bm.report('unrefined') {
str = 'foo'
1_000_000.times {
str.blah2
}
}
```

```
bm.report('refined') {
  using X
  str = 'foo'
  1_000_000.times {
    str.blah
  }
}
```

```
}
}
```

Best results for JRuby and MRI trunk:

JRuby:

```
Rehearsal ---------------------------------------------
unrefined   0.100000   0.000000   0.100000 (  0.095000)
refined     0.100000   0.000000   0.100000 (  0.099000)
------------------------------------ total: 0.200000sec
```

```
            user     system      total        real
```

```
unrefined   0.100000   0.000000   0.100000 (  0.094000)
```

refined   0.100000   0.000000   0.100000 (0.098000)

MRI:

Rehearsal --
unrefined   0.070000   0.000000   0.070000 (0.076026)
refined   0.080000   0.000000   0.080000 (0.076520)
----------------------------------- total: 0.150000sec

```
            user     system      total        real
```

unrefined   0.080000   0.000000   0.080000 (0.078869)
refined   0.080000   0.000000   0.080000 (0.077133)

JRuby compiled performance should be significantly better, but this at least shows there's not a great cost at the refined call sites.

#159 - 11/22/2012 08:23 AM - The8472 (Aaron G)

On 21.11.2012 07:15, shugo (Shugo Maeda) wrote:

> I'll remove it if permission granted by Matz.
>
>> For now people can use Module.extended/.included if they really want to
>> add refinement inheritance themselves.
>
> Currently this wouldn't work because you cannot get the caller context in these hooks.

What about the following?

```
module RefinementInheritor
  def extended(base)
    base.send(:using, FooExt)
    # or
    base.module_eval "using FooExt"
  end
end
```

>> Probably the safest approach for now would be to use the source
>> refinement scope (which is quasi-static) for module_eval by default and
>> add a way to use the target scope (or an explicit scope) later on as
>> needed. If there is any performance impact it would restricted to the
>> target-scoped procs.
>
> Do you mean that a new option of module_eval should be introduced?
> For example,
>
> Foo.module_eval { # use refinements in the current context }
> Foo.module_eval(using_refinements: true) { # use refinements in the receiver }

I was thinking about

```
Proc.new.rebind_refinements(TargetClass)
```

since this would only allow a single scope per proc at any given time
which might make optimizations easier. But maybe your way would work too.

> Originally, String#bar was not visible in the Proc created by Symbol#to_proc.
> But I've changed it because Matz asked to do. I think the current behavior
> is not consistent, but useful.
>
> If Symbol#to_proc were written in Ruby, it would be impossible, but
> Symbol#to_proc is written in C. There are some such special methods.
> For example, Module.nesting returns the module nesting information in the
> caller context. Module#using also affects the caller context.

So we need to special-case .to_proc. What happens when I
alias-method-chain to_proc? Would it use the wrong scope? Would things
break?

.**send** and .method obviously suffer from the same issues of shifting

stack frames and aliasing. Other metaprogramming things might be
expected to do the "right thing(tm)" too and thus would have to rely on
stack inspection which is an absolute minefield in Ruby.

Should the anonymous refinement module be mutable? E.g. by adding some
respond_to_missing to it?

Has anyone even defined how metaprogramming should work with refinements?

#160 - 11/22/2012 02:40 PM - shugo (Shugo Maeda)

The8472 (Aaron G) wrote:

> On 21.11.2012 07:15, shugo (Shugo Maeda) wrote:
>
> > I'll remove it if permission granted by Matz.
> >
> > > For now people can use Module.extended/.included if they really want to
> > > add refinement inheritance themselves.
> >
> > Currently this wouldn't work because you cannot get the caller context in these hooks.
>
> What about the following?
>
> ```
> module RefinementInheritor
>   def extended(base)
>     base.send(:using, FooExt)
>     # or
>     base.module_eval "using FooExt"
>   end
> end
> ```

The above code cannot activate refinements in the caller context.
That is, a NoMethodError is raised in the following example:

module FooExt
refine String do
def foo
puts "foo"
end
end
end

module RefinementInheritor
def self.included(mod)
mod.send(:using, FooExt)
# or
mod.module_eval "using FooExt"
end
end

module Foo
include RefinementInheritor

p "abc".foo #=> NoMethodError
end

> I was thinking about
>
> ```
> Proc.new.rebind_refinements(TargetClass)
> ```
>
> since this would only allow a single scope per proc at any given time
> which might make optimizations easier. But maybe your way would work too.

What happens if the receiver of rebind_refinements has already been called before rebind_refinements?

p = Proc.new { ... }
p.call
p.rebind_refinements(TargetClass)
p.call

> > Originally, String#bar was not visible in the Proc created by Symbol#to_proc.
> > But I've changed it because Matz asked to do. I think the current behavior
> > is not consistent, but useful.
> >
> > If Symbol#to_proc were written in Ruby, it would be impossible, but
> > Symbol#to_proc is written in C. There are some such special methods.
> > For example, Module.nesting returns the module nesting information in the
> > caller context. Module#using also affects the caller context.
>
> So we need to special-case .to_proc. What happens when I
> alias-method-chain to_proc? Would it use the wrong scope? Would things
> break?

Do you mean to redefine Symbol#to_proc yourself? If so, it's impossible to close refinements in the caller context of Symbol#to_proc into the created Proc.

> .**send** and .method obviously suffer from the same issues of shifting
> stack frames and aliasing. Other metaprogramming things might be
> expected to do the "right thing(tm)" too and thus would have to rely on
> stack inspection which is an absolute minefield in Ruby.

send and .method work the same as Symbol#to_proc in the current implementation.

> Should the anonymous refinement module be mutable? E.g. by adding some
> respond_to_missing to it?

Currently respond_to_missing doesn't work in a refinement module.

module FooExt
refine String do
def respond_to_missing?(mid, include_all)
mid == :foo
end

```
def method_missing(mid, *args)
  if mid == :foo
    puts "foo!"
  else
    super
  end
end
```

end
end

using FooExt

if "abc".respond_to?(:foo) #=> false
"abc".foo
end

I guess respond_to? need to be fixed to make the above code work.

> Has anyone even defined how metaprogramming should work with refinements?

I think, in principle, metaprogramming APIs related to method dispatching should use refinements in the caller context.

#161 - 11/22/2012 02:56 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> I would also vote to remove refinements from 2.0 features, perhaps for reinclusion in 2.1. The various points made about lack of clear specification, lack of time to experiment, lack of clarity on the extent of damage/risk, and lack of time for other implementers to fully implement and explore the feature all point toward this being too much, too late in 2.0's dev cycle...and I believe it is too close to 2.0's release to shove the features in.

It may be better to remove Refinements completely than to introduce Refinements partially.
I defer all decisions to Matz.

> That said, I will say I appreciate the reduction in scope. Having refinements only searched via lexical enclosures makes the feature much simpler to implement and much easier to understand in real code. To help the process of fleshing out the feature, I have implemented a large part of the reduced refinements feature for JRuby on the refinements branch: https://github.com/jruby/jruby/tree/refinements

Thank you. I've tried it and it works well as a lexical version of Refinements.

#162 - 11/22/2012 09:07 PM - jballanc (Joshua Ballanco)

shugo (Shugo Maeda) wrote:

> headius (Charles Nutter) wrote:
>
>> I would also vote to remove refinements from 2.0 features, perhaps for reinclusion in 2.1. The various points made about lack of clear specification, lack of time to experiment, lack of clarity on the extent of damage/risk, and lack of time for other implementers to fully implement and explore the feature all point toward this being too much, too late in 2.0's dev cycle...and I believe it is too close to 2.0's release to shove the features in.
>
> It may be better to remove Refinements completely than to introduce Refinements partially.
> I defer all decisions to Matz.

I wonder if refinements could be included as a "hidden" feature, similar to tail-call optimization? I suspect that even if refinements were included in Ruby 2.0 it would be at least a year or two before library authors (presumably the true audience for this feature) could really make use of refinements, as they will likely want to continue supporting Ruby 1.9 in the near term. I think leaving them in but "off" by default would allow library authors to begin experimenting with refinements, find all of the various corner cases, and then there could be a well defined roadmap for resolving the remaining issues and having refinements enabled by default.

#163 - 11/23/2012 01:40 AM - headius (Charles Nutter)

Escaping the "should we or shouldn't we" question for a bit, I thought of an alternative implementation, building off ko1's idea.

ko1's suggestion, as I understand it, was to add a flag to the method table (or method entry) of a refined class/method as a trigger for the call site to search refinements. While writing my blog post, I started to type the sentence "the methods defined in the refinement do not actually go on the class in question", and then I realized: why not?

Currently, Ruby implementations structure the method table as a simple map from names to method bodies. If instead the method table was a map from names to collections of methods, we could use that to choose the appropriate method for a given context.

So, for code like this:

```
module X
refine String do
def upcase; downcase; end
end
end
```

String's method table would contain an entry like this:

```
{:upcase => {
:default => ,
X => }
}
```

Method lookup would then proceed as normal in all situations. The result of lookup would be a table mapping refinements to methods with a default entry if the method is defined directly on String.

After lookup, call sites would know there's potentially refinements active for the given method. The calling scope (or parent scopes) would have references to individual refinements, and if there were an entry for one of them it would be used.

This still requires access to the caller scope, of course, to understand what refinements are active. However, because refinement changes would invalidate the String class directly (since they actually modify the method table), the method (refined or otherwise) could be cached as normal. The caller's scope never changes (statically determined at compile time), so it does not participate in invalidation.

This also works for refinements added to classes after the initial run through. If we cache the default downcase method from String, and then the refinement is updated to add downcase, we would see that as an invalidation event for String's method table. Future calls would then re-cache and pick up the change.

This also feels a bit more OO-friendly to me. Rather than storing patches on separate structures sprinkled around memory, we store the patches directly on the refined class, only using the module containing the refinements as a key. The methods *do* live on String, but depending on the *namespace* they're looked up from we *select* the appropriate implementation. It's basically just double-dispatch at that point, with the selector being the calling context.

It also makes available an interesting possibility for #method and friends: return all methods. So...

```
using X

String.instance_methods(:upcase) # => {:default => , X => }
```

Note that this is "instance_methods", plural, to avoid breaking instance_method and to make it explicit that we're asking for all implementations of a given method. This allows accessing the original method even if refinements are active, and still also allows searching for the refined method active in the current scope.

I admit I am a bit reluctant to suggest this, because I still have concerns about the feature itself. But it would be possible for call sites to only need a reference to their calling scope (determined at parse time) to implement dynamic refinements without severe impact to normal code. Dynamic refinements, as in module_eval, would work by simply invalidating the call sites they contain. This could be done actively, walking all call sites and resetting them. This could also be done by invalidating the classes refined. An example in pseudo-code:

```
def X.module_eval_refined(&block)
unless block.using? self
refinements.each_key {|cls| cls.touch } # invalidate all refined classes
block.using(self)
end
module_eval &block
end
```

This is obviously not a thread-safe mechanism. An alternative that invalidates the block's call sites (this would require more work and be more expensive at invalidation time, but less globally-damaging):

```
def X.module_eval_refined(&block)
unless block.using? self
block.invalidate
block.using(self)
end
module_eval &block
end
```

Proc#using would either mutate the block's already-present scope (permanently adding the refinement) or duplicate the block and its scope and tweak it (more expensive, of course).

In any case, I would really like more time for this dialog to continue. If we push refinements into Ruby in their current form, we're not giving adequate time to flesh out the edge cases. If we push a partial implementation now, we may be making a future implementation harder and we would not be protecting ourselves from mistakes. I want to work with you to find a definition and implementation of refinements that meets requirements without punishing future Rubyists.

I also must apologize for not joining the dialog sooner. This bug was filed in 2010, and the current refinements implementation was pushed to master a few months ago. We should have started discussing a long time ago.

#164 - 11/23/2012 05:59 AM - The8472 (Aaron G)

On 22.11.2012 17:40, headius (Charles Nutter) wrote:

> After lookup, call sites would know there's potentially refinements active for the given method. The calling scope (or parent scopes) would have references to individual refinements, and if there were an entry for one of them it would be used.
>
> This still requires access to the caller scope, of course, to understand what refinements are active. However, because refinement changes would invalidate the String class directly (since they actually modify the method table), the method (refined or otherwise) could be cached as normal. The caller's scope never changes (statically determined at compile time), so it does not participate in invalidation.
>
> This also works for refinements added to classes after the initial run through. If we cache the default downcase method from String, and then the refinement is updated to add downcase, we would see that as an invalidation event for String's method table. Future calls would then re-cache and pick up the change.
>
> This also feels a bit more OO-friendly to me. Rather than storing patches on separate structures sprinkled around memory, we store the patches directly on the refined class, only using the module containing the refinements as a key. The methods *do* live on String, but depending on the *namespace* they're looked up from we *select* the appropriate implementation. It's basically just double-dispatch at that point, with the selector being the calling context.

This (together with Module.prepend) is reminding me a bit of AspectJ's
pointcuts. Which in turn leads me to think that we are missing something
here:
We don't know and cannot know in advance which kind of scopes the
developer will need to apply his patches.

We have many different ideas flying around how to determine the scope of
the refinement.

a) Local only? Maybe even constrained inside a block?

Good for builder DSLs or the like where you basically want to extend
core objects to make it look more like written sentences than code.

b) Class inheritance?

E.g. If you want to provide some nice class configuration syntax

c) Module namespace?

If you like to use some convenience methods throughout your project
without creating conflicts with extensions that libraries might use in
their own code

d) Stack-down X frames

Black Magic: Patch some behavior inside a single method by wrapping it

e) Thread local

More black magic: Fix some broken interaction between library code.
Stub any kind of method out temporarily without breaking other things in
multi-threaded environments.

So what I am saying is that we don't just need a way to define
refinement namespaces. We also need to let the programmer define where
and when those namespaces get applied. And we need the common cases to
be fast. The madness-driven ones (d and e) can be slow, but can only be
allowed to be slow at those callsites that are affected, not globally.

So I would suggest not providing *any* inheritance at all. Refinements
scopes must be activated in every single module (or possibly even
method) that they should be applied to. They shouldn't even apply to
methods overriding another method when the super-method is
refinement-scoped. If you want to apply them in many places at once you
can do so via metaprogramming.

```
module FooExt
  refine String do
    def downcase
      upcase
    end
  end
end
```

case a)

```
class ClassA
  def bar; end

  # apply to all methods when no block is passed
  using(FooExt)

  def baz; end

  # both .bar and .baz are refined now.
  # we can apply them retroactively!
  # this is important for monkey-patching
end

class ClassB
  def foo
    "x".downcase => "x"
    using_refinements(FooExt) do
      "x".downcase => "X"
    end
  end
end

class ClassC
  def bar
    "x".downcase
  end
  def baz
    "x".downcase
  end
```

```
end

o = ClassC.new

o.bar # => "x"
o.send(:using, FooExt, :on => :bar)
o.bar # => "X"
o.baz # => "x"

# acts as instance_eval with refinements
Object.new.using_refinements(FooExt) do
  "x".downcase # => "X"
end
```

case b)

```
class MyModel < ActiveRecord::Base; end
class SubModel < MyModel; end
class OtherModel < ActiveRecord::Base; end

ActiceRecord::Base.descendants.each do |c|
  c.send(:using, FooExt)
end
```

case c)

```
# assume this traverses the constants downwards
MyApplicationNamespace.recursive_submodules.each do |mod|
  mod.send(:using, FooExt)
end

# only modify callsites for a single method
Bar.instance_method(:test).using(FooExt)
```

case d)
case e)

```
# applies refinement to callsites in method :bar in MyClass
# but only if the guard condition is true
# otherwise the unrefined method is used
MyClass.send(:using, FooExt, :on => :bar, :if => lambda do
   Thread[:use_string_patches?]
end)

# applies FooExt a dynamic refinement scope
# to *all* String.downcase callsites throughout the application
FooExt.send(:use_everywhere) do
   Thread[:use_string_patches?] && caller[1]["<stack match here>"]
end
```

I think this should demonstrate the power of letting the programmer
decide how refinement scopes are determined instead of having the
language dictate a fixed lookup strategy.

Cases d) and e) are just for demonstration and don't have to be taken
seriously!

But the metaprogramming issues with **send**, respond_to? and
Symbol.to_proc would still remain.

#165 - 11/24/2012 02:04 PM - mame (Yusuke Endoh)

To be honest, I do not follow this discussion at all. I just heard from ko1 that there are still room to discuss this feature.

I propose: will we release the refinement as an "experimental" feature? It is enough for 2.0 to clarify:

- how usage is "defined" (we will ensure the compatibility), and
- how usage is "undefined" (the behavior may change in future)

By doing so, we will be able to improve (or refine) the feature based on actual experiment in 2.0.0.

(I'll come back to this ticket to understand the discussion on a deeper level, after I finished other tasks.
I will be very very happy to create a summary of the discussion.)

--

Yusuke Endoh mame@tsg.ne.jp

#166 - 11/24/2012 02:07 PM - mame (Yusuke Endoh)

P.S. This is just my current impression, but we have no option to remove the whole feature from 2.0.0.

--
Yusuke Endoh mame@tsg.ne.jp

#167 - 11/24/2012 02:21 PM - duerst (Martin Dürst)

mame (Yusuke Endoh) wrote:

> I propose: will we release the refinement as an "experimental" feature? It is enough for 2.0 to clarify:
>
> - how usage is "defined" (we will ensure the compatibility), and
> - how usage is "undefined" (the behavior may change in future)
>
> By doing so, we will be able to improve (or refine) the feature based on actual experiment in 2.0.0.

If this means that every time (or the first time) refinements are used in a program, a warning is issued, e.g. "Use of refinements is currently experimental, and implementation may change in future versions of Ruby!", then this would be okay with me. If there is no such warning, I'm affraid that we won't be able to make changes even if we want.

#168 - 11/24/2012 02:58 PM - mame (Yusuke Endoh)

duerst (Martin Dürst) wrote:

> If this means that every time (or the first time) refinements are used in a program, a warning is issued, e.g. "Use of refinements is currently experimental, and implementation may change in future versions of Ruby!", then this would be okay with me. If there is no such warning, I'm affraid that we won't be able to make changes even if we want.

Ideally, only "undefined" behavior should be warned.
If they are automatically indistinguishable,,, it is unfortunate.

Anyway I should study this long long discussion...

--
Yusuke Endoh mame@tsg.ne.jp

#169 - 11/24/2012 06:10 PM - dbussink (Dirkjan Bussink)

Here are few points I'd like to make as to why I think the feature should not be part of 2.0. First of all, there is currently not a single idea / design that describes what refinements are. In this issue different approaches and ideas are explored. This is of course a good thing, but it also means that we haven't come to a conclusion.

This means that if refinements were to be part of Ruby 2.0, they would be an implementation that hasn't been fleshed out, is still being debated and can change in the future. I think doing this would be a great disservice to the Ruby community. This is because we would release a feature that people cannot depend on. It may change in the future, break people's code and cause heavy debates on whether we can actually change it in the future.

As a reference, originally in 1.9, Hash being insertion ordered was defined as an implementation detail of CRuby. The position of CRuby in the community however, means that this also turned into spec since people depend on it. This was basically a decision made for us by the community, neither JRuby or Rubinius could say, for us it's not insertion ordered. This means that if some things are released, what is and what is not defined behavior is not always just up to the implementers. The Ruby community will come to expect certain behavior of certain features and depend on it.

If Ruby 2.0 is released without a properly defined way of how refinements work and that they will be supported in that way well into the future, we very well may end up with an implementation that no one really likes but has to be supported indefinitely.

Ruby is not an experimental language anymore, so asking of the community to accept that this all can change in the future is not the right thing to do. What is part of Ruby the language is something that we should be able to stand behind and say of "we will support this and provide a stable platform for you for the future". If that can't be said of a feature, it should not be added to a released Ruby version.

Even if removing the feature for now is a hard decision, I feel it is the correct. We should not fall into a sunk cost fallacy, where the amount of development on it results in feeling like it could not be removed.

Please realize that nowhere in the argument I've discussed my personal preferences on the feature itself. The thing is that I don't think that should be even necessary at this point, because of all the unclarity about the feature and how it should work.

All in all, I think the Ruby community would be done a great disservice by adding a feature that is still being debated so short before a 2.0 release, does not have clear semantics (looking at the alternatives discussed here) and therefore isn't mature enough to be put out there in the wild.

#170 - 11/26/2012 05:48 AM - trans (Thomas Sawyer)

I was reading this discussion (http://branch.com/b/rubyists-which-would-you-rather-have-in-ruby-2-0-0) about refinements, and read the last post by Magnus Holm (judofyr (Magnus Holm)) which said:

> I think refinements are useful for some specific tasks; String#camelize is *not* one of them.
> I see no real gain of using "foo".camelize over StringUtils.camelize("foo").
>
> No, I think refinements should be used for DSLs. As wycats (Yehuda Katz) mentioned: It's the difference
> between DataMapper.not(:age => 12) and :age.not => 12. Or 5.days.ago instead
> of DateUtilts.now.subtract(:days => 5).
>
> And in that context I think that *lexical* refinements makes a whole lot more sense than
> dynamic refinements; both in terms of debugging and in terms of sanity.

For a while I too was thinking that same thing. But now it seems to me that perhaps this kind of DSL is just bad API design --in particular the whole idea of adding a bunch of methods to Symbol class.

For example, something like :age.not => 12 could be written as a block DSL instead as { age != 12 }. That's so much sexier anyway.

As for something like 5.days.ago, I agree with Steve Klabnik who calls such cases more of a "social" problem. It is something that's common enough that it would be better if Ruby had a concept of Duration built-in and some convenient conversion methods on Numeric. At worse there could be single entry point like 5.calendar.days.ago or maybe 5.th(:day).ago. Certainly we don't need something as course as DateUtilts.now.subtract(:days => 5), but we could still do something reasonable without a bunch of refinements.

So I guess my point is that refinements are not really even addressing a "20%" problem, it seems more like a "1%" problem. And as cool as they seem, I am not so sure that's enough.

Again, I'll mention that I am a bit afraid of what might happen if refinements become popular. Imagine a library with dozens of various refinements on all the core classes. Might we start seeing DSLs like: :batter.count(1.ball && 2.strikes).

#171 - 11/26/2012 06:34 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> ko1's suggestion, as I understand it, was to add a flag to the method table (or method entry) of a refined class/method as a trigger for the call site to search refinements. While writing my blog post, I started to type the sentence "the methods defined in the refinement do not actually go on the class in question", and then I realized: why not?
>
> Currently, Ruby implementations structure the method table as a simple map from names to method bodies. If instead the method table was a map from names to collections of methods, we could use that to choose the appropriate method for a given context.
>
> So, for code like this:
>
> module X
> refine String do
> def upcase; downcase; end
> end
> end
>
> String's method table would contain an entry like this:
>
> {:upcase => {
> :default => ,
> X => }
> }
>
> Method lookup would then proceed as normal in all situations. The result of lookup would be a table mapping refinements to methods with a default entry if the method is defined directly on String.

It's an interesting idea.
How does method lookup work if multiple modules are used by using?

module Z
using X
using Y
"foo".upcase
end

Are both X and Y used as a key of the table in the reverse order they are used by using?

And, how does super work?

> I admit I am a bit reluctant to suggest this, because I still have concerns about the feature itself. But it would be possible for call sites to only need a reference to their calling scope (determined at parse time) to implement dynamic refinements without severe impact to normal code.

> Dynamic refinements, as in module_eval, would work by simply invalidating the call sites they contain.

FYI, in my new implementation (http://shugo.net/tmp/refinement_fix_1119.diff), refined methods are not stored in inline cache, so there's no need to invalidate inline cache for module_eval.
Instead, refined method invocations are slower than the implementation in the trunk HEAD.

> In any case, I would really like more time for this dialog to continue. If we push refinements into Ruby in their current form, we're not giving adequate time to flesh out the edge cases. If we push a partial implementation now, we may be making a future implementation harder and we would not be protecting ourselves from mistakes. I want to work with you to find a definition and implementation of refinements that meets requirements without punishing future Rubyists.

I'm starting to think it's not good to rush to introduce Refinements as an official feature.
What do you think of introducing Refinements as an experimental feature as Endoh-san suggested?
I don't know what does Endoh-san mean by "an experimental feature", but it may require an explicit compile option or a runtime option (e.g., require "refinements" like continuation) to enable it. If Refinements are enabled, warning should be shown not to make the current Refinements de-facto standard.

> I also must apologize for not joining the dialog sooner. This bug was filed in 2010, and the current refinements implementation was pushed to master a few months ago. We should have started discussing a long time ago.

I should also have warned people to think of Refinements sooner. Anyway, thanks for your comments.

#172 - 11/27/2012 01:37 AM - headius (Charles Nutter)

shugo (Shugo Maeda) wrote:

> headius (Charles Nutter) wrote:
>
>> {:upcase => {
>> :default => ,
>> X => }
>> }
>>
>> Method lookup would then proceed as normal in all situations. The result of lookup would be a table mapping refinements to methods with a default entry if the method is defined directly on String.
>
> It's an interesting idea.
> How does method lookup work if multiple modules are used by using?
>
> module Z
> using X
> using Y
> "foo".upcase
> end
>
> Are both X and Y used as a key of the table in the reverse order they are used by using?

I would think they stack like module includes, so at lookup time we'd see refined methods on String, look in calling scope in reverse order, and use the first refinement we encounter as the key.

> And, how does super work?

Well, I'm still questioning how super should work in general. Refinements are not actually modifying class hierarchy, so the current behavior of super calling the old method seems like magic to me.

You have it implemented currently as though used refinements are "virtually" in the hierarchy beneath the class, similar to prepend. So stacked refinements fire "super" in reverse order:

```
irb(main):001:0> module X; refine(String) { def upcase; puts 'first'; super; end }; end
=> #Module:0x007ffbf30a4898
irb(main):002:0> module Y; refine(String) { def upcase; puts 'second'; super; end }; end
=> #Module:0x007ffbf3082ae0
irb(main):003:0> using X
=> main
irb(main):004:0> using Y
=> main
irb(main):005:0> 'foo'.upcase
second
first
```

```
=> "FOO"
```

But there are some oddities when refinements are mixed into multiple elements of the hierarchy:

```
irb(main):001:0> module A; refine(Numeric) { def blah; puts 'A'; super; end }; end
=> #Module:0x007ffeeb930d90
irb(main):002:0> module B; refine(Integer) { def blah; puts 'B'; super; end }; end
=> #Module:0x007ffeeb90f578
irb(main):003:0> module C; refine(Fixnum) { def blah; puts 'C'; super; end }; end
=> #Module:0x007ffeeb8f1870
irb(main):004:0> using A; using B; using C
=> main
irb(main):005:0> 1.blah
C
NoMethodError: super: no superclass method blah' for 1:Fixnum
from (irb):3:inblah'
from (irb):6
from /usr/local/bin/irb-2.0.0:12:in `'
```

Obviously refined super is not simulating the full hierarchy here. It would be difficult to do so, but I feel like you either need to support super in refinements consistently or not at all.

If I add modules to the same locations in the hierarchy, the supers fire "properly". So basically, refined "super" is only working for one level up from the refinement itself.

```
irb(main):007:0> module A2; def blah; puts 'A2'; super; end; end
=> nil
irb(main):008:0> module B2; def blah; puts 'B2'; super; end; end
=> nil
irb(main):009:0> module C2; def blah; puts 'C2'; super; end; end
=> nil
irb(main):010:0> class Fixnum; prepend C2; end
=> Fixnum
irb(main):011:0> class Integer; prepend B2; end
=> Integer
irb(main):012:0> class Numeric; prepend A2; end
=> Numeric
irb(main):013:0> 1.blah
C
C2
C
B2
B
A2
A
NoMethodError: super: no superclass method blah' for 1:Fixnum
from (irb):1:inblah'
from (irb):7:in blah'
from (irb):2:inblah'
from (irb):8:in blah'
from (irb):3:inblah'
from (irb):9:in blah'
from (irb):3:inblah'
from (irb):13
from /usr/local/bin/irb-2.0.0:12:in `'
```

The double call of C's blah here is unexpected as well.

Another example showing that refinements don't honor refined hierarchies for "super":

```
irb(main):026:0> class Foo
irb(main):027:1> def blah; puts 'in Foo'; end
irb(main):028:1> end
=> nil
irb(main):029:0> class Bar < Foo
irb(main):030:1> def blah; puts 'in Bar'; super; end
irb(main):031:1> end
=> nil
irb(main):032:0> module Baz
irb(main):033:1> refine Foo do
irb(main):034:2* def blah; puts 'in Baz'; super; end
irb(main):035:2> end
irb(main):036:1> end
=> #Module:0x007ffeeb05f978
irb(main):037:0> using Baz
```

```
=> main
irb(main):038:0> Bar.new.blah
in Bar
in Foo
=> nil
```

Again, inconsistent behavior, but I'm not sure which specification is correct.

FWIW, there's something similar to refinements in the form of extension methods in C# and defender methods in Java 8. It would be worth researching how those features handle super. In Java 8, the defender methods live only on interfaces and are somewhat "virtually" in the hierarchy, so there's a lot of oddities surrounding the process of selecting the proper super method.

> > I admit I am a bit reluctant to suggest this, because I still have concerns about the feature itself. But it would be possible for call sites to only need a reference to their calling scope (determined at parse time) to implement dynamic refinements without severe impact to normal code. Dynamic refinements, as in module_eval, would work by simply invalidating the call sites they contain.
>
> FYI, in my new implementation (http://shugo.net/tmp/refinement_fix_1119.diff), refined methods are not stored in inline cache, so there's no need to invalidate inline cache for module_eval.
> Instead, refined method invocations are slower than the implementation in the trunk HEAD.

I considered this possibility, but are you willing to accept that large parts of Rails code will have slower overall performance because they want to use refinements? I will revise my earlier refinements requirements: refined calls should exhibit exactly the same performance characteristics as regular calls. I believe if refinements go in, many many libraries will want to start using them. We should not force Ruby perf to take a major step backward just by introducing a new and potentially popular feature that has implementation problems.

> > In any case, I would really like more time for this dialog to continue. If we push refinements into Ruby in their current form, we're not giving adequate time to flesh out the edge cases. If we push a partial implementation now, we may be making a future implementation harder and we would not be protecting ourselves from mistakes. I want to work with you to find a definition and implementation of refinements that meets requirements without punishing future Rubyists.
>
> I'm starting to think it's not good to rush to introduce Refinements as an official feature.
> What do you think of introducing Refinements as an experimental feature as Endoh-san suggested?
> I don't know what does Endoh-san mean by "an experimental feature", but it may require an explicit compile option or a runtime option (e.g., require "refinements" like continuation) to enable it. If Refinements are enabled, warning should be shown not to make the current Refinements de-facto standard.

I do not have any objection to refinements being included as an experimental feature.

If it's a compile-time feature, I'm not sure I see the value in having it in 2.0.0 at all; people could download source and build that.

If it's a flag or require, I assume you'd have to enable it to turn on parse/compile-time flagging of refined methods/calls, correct? I think that's easy enough in JRuby too.

#173 - 11/27/2012 06:24 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> I would think they stack like module includes, so at lookup time we'd see refined methods on String, look in calling scope in reverse order, and use the first refinement we encounter as the key.

I see.

> > And, how does super work?
>
> Well, I'm still questioning how super should work in general. Refinements are not actually modifying class hierarchy, so the current behavior of super calling the old method seems like magic to me.

It may seem magical, but is intended for use like aspect oriented programming.

> But there are some oddities when refinements are mixed into multiple elements of the hierarchy:
>
> ```
> irb(main):001:0> module A; refine(Numeric) { def blah; puts 'A'; super; end }; end
> => #Module:0x007ffeeb930d90
> irb(main):002:0> module B; refine(Integer) { def blah; puts 'B'; super; end }; end
> => #Module:0x007ffeeb90f578
> irb(main):003:0> module C; refine(Fixnum) { def blah; puts 'C'; super; end }; end
> => #Module:0x007ffeeb8f1870
> irb(main):004:0> using A; using B; using C
> ```

```
=> main
irb(main):005:0> 1.blah
C
NoMethodError: super: no superclass method blah' for 1:Fixnum
from (irb):3:inblah'
from (irb):6
from /usr/local/bin/irb-2.0.0:12:in `'
```

> Obviously refined super is not simulating the full hierarchy here. It would be difficult to do so, but I feel like you either need to support super in refinements consistently or not at all.

I admit that the above example looks odd. I'd like to fix it if possible.

> The double call of C's blah here is unexpected as well.

It looks odd too.

> Another example showing that refinements don't honor refined hierarchies for "super":
>
> ```
> irb(main):026:0> class Foo
> irb(main):027:1> def blah; puts 'in Foo'; end
> irb(main):028:1> end
> => nil
> irb(main):029:0> class Bar < Foo
> irb(main):030:1> def blah; puts 'in Bar'; super; end
> irb(main):031:1> end
> => nil
> irb(main):032:0> module Baz
> irb(main):033:1> refine Foo do
> irb(main):034:2* def blah; puts 'in Baz'; super; end
> irb(main):035:2> end
> irb(main):036:1> end
> => #Module:0x007ffeeb05f978
> irb(main):037:0> using Baz
> => main
> irb(main):038:0> Bar.new.blah
> in Bar
> in Foo
> => nil
> ```
>
> Again, inconsistent behavior, but I'm not sure which specification is correct.

I and Matz discussed this behavior before, and concluded that Baz's blah should not be called in this case, because Bar's blah is outside the scope where Baz is activated and Refinements do not support local rebinding.
However, I admit that the behavior looks odd.

> FWIW, there's something similar to refinements in the form of extension methods in C# and defender methods in Java 8. It would be worth researching how those features handle super. In Java 8, the defender methods live only on interfaces and are somewhat "virtually" in the hierarchy, so there's a lot of oddities surrounding the process of selecting the proper super method.

I'll check them. Thank you.

> > > I admit I am a bit reluctant to suggest this, because I still have concerns about the feature itself. But it would be possible for call sites to only need a reference to their calling scope (determined at parse time) to implement dynamic refinements without severe impact to normal code. Dynamic refinements, as in module_eval, would work by simply invalidating the call sites they contain.
> >
> > FYI, in my new implementation (http://shugo.net/tmp/refinement_fix_1119.diff), refined methods are not stored in inline cache, so there's no need to invalidate inline cache for module_eval.
> > Instead, refined method invocations are slower than the implementation in the trunk HEAD.
>
> I considered this possibility, but are you willing to accept that large parts of Rails code will have slower overall performance because they want to use refinements? I will revise my earlier refinements requirements: refined calls should exhibit exactly the same performance characteristics as regular calls. I believe if refinements go in, many many libraries will want to start using them. We should not force Ruby perf to take a major step backward just by introducing a new and potentially popular feature that has implementation problems.

I think the performance of refined calls can be improved by new cache dedicated for refinements, but it might still be a little slower than normal calls.

> I do not have any objection to refinements being included as an experimental feature.

> If it's a compile-time feature, I'm not sure I see the value in having it in 2.0.0 at all; people could download source and build that.

It may not be worth having such a feature. If people can build it themselves, they can use SVN trunk.

> If it's a flag or require, I assume you'd have to enable it to turn on parse/compile-time flagging of refined methods/calls, correct? I think that's easy enough in JRuby too.

I meant to provide refinements.so, which just publishes Module#refine, Module#using, etc... in Ruby level, like continuation.so.

#174 - 11/28/2012 05:59 AM - headius (Charles Nutter)

shugo (Shugo Maeda) wrote:

> headius (Charles Nutter) wrote:
>
> > Well, I'm still questioning how super should work in general. Refinements are not actually modifying class hierarchy, so the current behavior of super calling the old method seems like magic to me.
>
> It may seem magical, but is intended for use like aspect oriented programming.

The interaction between super chains and refinements bothers me.

A new idea...

Because we'd like refinements to act like they live in the class hierarchy...let's just make them live in the class hierarchy.

So...we start out with String < Object. Do a refinement:

module X
refine String do
def blah; end
end
end

Now the String hierarchy looks like this: RefinedByX < String < Object

It's rather prepend-like. Lookup proceeds as normal for a given string by getting the metaclass (now RefinedByX). However when refined intermediate classes are encountered, the calling scope is queried to see if it has activated that refinement.

CallSite pseudo-code:

metaclass = obj.metaclass
while metaclass != null
if metaclass.refinement?
unless caller_scope.include_refinement? metaclass
# skip refinement
metaclass = metaclass.superclass
next
end
end

method = metaclass.search_method method_name
return method if method
metaclass = metaclass.superclass
end

Super logic than can operate as it should, using the refined scope to do the subsequent super lookup, which it finds in the hierarchy in the normal way. Caching also proceeds largely the same, based on the target object's hierarchy only; the only difference is whether refined elements in the hierarchy are included in the search.

Not sure about other edge cases for this, but it's another way to look at it, and I think this starts to move refinements more directly toward being structured in the same way as module inclusion rather than based on the more magical concept of "current frame overlay modules".

#175 - 11/29/2012 01:56 PM - matz (Yukihiro Matsumoto)

Since there still remain undefined corner case behavior in refinements, and the time is running out, I decided not to introduce full refinement for Ruby 2.0. The limited Ruby 2.0 refinement spec will be:

- refinements are file scope
- only top-level "using" is available
- no module scope refinement

- no refinement inheritance
- module_eval do not introduce refinement (even for string args)

In addition, Module#include should add refinements to included modules, e.g.

module R1
refine String do
def bar
p :bar
end
end
end

module R2
include R1
refine String do
def foo
p :foo
end
end
end

using R2
"".foo
"".bar

module R1
refine String do
def bar; p :bar end
end
end

module R2
include R1
refine String do
def foo; p :foo end
end
end

using R2
"".foo
"".bar # does not work now

You can treat top-level "using" as soft-keyword, as long as it does not change the behavior (but performance).

Matz.

#176 - 11/29/2012 03:02 PM - shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

> Since there still remain undefined corner case behavior in refinements, and the time is running out, I decided not to introduce full refinement for Ruby 2.0. The limited Ruby 2.0 refinement spec will be:

I don't understand what do you mean by these constraints. Let me ask some questions.

> - refinements are file scope
> - only top-level "using" is available
> - no module scope refinement

Do these constraints just mean that main.using is available, but Module#using is not?
How should the following code behave?

module R
refine String do
def foo; p :foo; end
end
"".foo # (a)
end
"".foo   # (b)

Currently, (a) prints :foo, and (b) raises a NoMethodError.
And, how about the following example, where a nested module is defined?

```
module R
refine String do
def foo; p :foo; end
end

module M
"".foo
end
"".foo
end
"".foo
```

If the behavior in the new spec is the same as the current implementation,
I don't know well why Module#using should be removed.

> - no refinement inheritance
> - module_eval do not introduce refinement (even for string args)

I understand these.

> In addition, Module#include should add refinements to included modules, e.g.

This is very different from the current feature, so we need a discussion about it.
What does "add refinements" mean here?
There are two aspects about refinement addition. They are defined in modules by Module#refine, and activated in certain scopes by using.
Does "to add refinements" mean to define (or inherit indirectly) refinements in modules, or to activate refinements in modules, or both of them?

For example, how should the following code behave?

```
module R1
refine String do
def bar
p :bar
end
end
end

module R2
include R1
refine String do
def foo
p :foo
end
end
"".foo
"".bar
end
```

Finally, how super in refinements should behave in the new spec?

#177 - 11/29/2012 05:43 PM - dbussink (Dirkjan Bussink)

The last two comments here again confirm my view that we should not add refinements to Ruby 2.0. Apparently it is not even clear to the people developing Ruby itself what the feature means. If there is ambiguity and questions like this have to asked, how can we add this with a straight face?

I feel we should not treat Ruby as an experimental place where we can try stuff like this that affects the whole language. Please consider removing refinements for 2.0, since this discussions keeps confirming that the ideas are not clear and not fleshed out. Adding half done features at the language level mean a serious disservice to the Ruby community and does not clearly communicate the path to the future.

#178 - 11/29/2012 06:09 PM - Gibheer (Stefan Radomski)

Hello,

I followed the discussion of refinements not as close as most others commenting here, but the state I saw till now was a bit disturbing. As I understand it, there are some definition, implementation and performance problems in MRI or other ruby implementations, which have to be considered.

When I understood Matz correct, he wants to add refinements in a different way in MRI 2.0. As some library maintainers already said, that they want to use refinements because it helps them, I think that is not a helpful step. Refinements will change afterwards in behavior and cause incompatibilities for future versions of refinements and ruby implementations.

As you already put so much effort into it, it would be bad to see it split and cut down to a state, which does not represent the original idea behind it. Take a step back and don't include it in 2.0 but instead use the time to polish it, write specifications and tests. With that you can give developers something to fully understand the trade-offs/impact of refinements and the use-case for the feature. This can also help developers working on the various ruby implementations.

thank you

#179 - 11/29/2012 06:31 PM - trans (Thomas Sawyer)

=begin

    refinements are file scope

That's a very interesting and significant recalibration of refinements. On one hand it certainly simplifies the whole scoping issue. On the other, would it mean we would have to reiterate using for every file, i.e. we couldn't gain a using via require?

```
cat foo.rb
require 'facets'
using Facets

cat bar.rb
require 'foo.rb'
# Are we using Facets ?
```

I suppose the answer would have to be "no", otherwise anyone who required it would have to be using it too, which would defeat the point whole of refinements.

Unfortunately it is a little annoying to have to put using at the top of every file. Is it possible to determine the gem a file comes from? Is it conceivable to have something like:

GEM #=> 'foo-1.2.1'

If so then using could be tied per-gem rather then just per file.
=end

#180 - 11/29/2012 06:59 PM - matz (Yukihiro Matsumoto)

trans (Thomas Sawyer) I am thinking of combination of require and using, but I don't want to put half baked idea into Ruby 2.0.
So I leave it to the future.

Matz.

#181 - 11/29/2012 08:19 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

matz (Yukihiro Matsumoto), I like the suggestion of "using" being file scoped, although I don't understand why you want it to be applied to top-level object only... Would you mind in explaining why you think this is more interesting or would simplify things?

#182 - 11/29/2012 08:43 PM - brainopia (Ravil Bayramgalin)

Yeah, "using" at file scope will mean a bit of boilerplate to specify it across several files, but since there are no leaking abstractions I think new proposal answers all concerns about debugging and readability.

And in future releases refinements can be easily improved to incorporate other features while staying backward compatible. Well done!

#183 - 11/29/2012 09:56 PM - steveklabnik (Steve Klabnik)

Wasn't a 'feature freeze' declared on October 24th? Why are major additions to the language being modified post-freeze, so close to the deadline for an actual release?

#184 - 11/29/2012 09:56 PM - steveklabnik (Steve Klabnik)

=begin
Sorry, I didn't realize that the form was submitted because redmine is being a bit slow today. This comment was a duplicate of my previous one, please ignore this one.
=end

#185 - 11/29/2012 11:23 PM - Anonymous

In message "Re: [ruby-core:50299] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Thu, 29 Nov 2012 15:02:03 +0900, "shugo (Shugo Maeda)" redmine@ruby-lang.org writes:

|> * refinements are file scope
|> * only top-level "using" is available
|> * no module scope refinement
|

|Do these constraints just mean that main.using is available, but Module#using is not?

Yes, only main.using should be available. No Module#using (for 2.0).

```
|How should the following code behave?
|
|module R
|  refine String do
|    def foo; p :foo; end
|  end
|  "".foo # (a)
|end
|"".foo   # (b)
|
|Currently, (a) prints :foo, and (b) raises a NoMethodError.
```

Refinements will be available only from:

- the scope where refinements are added by calling "using"
- or inside of refine blocks

Inside of refine blocks (not whole module scope) might be
controversial, but I think it's OK to restrict refinements there. As
a result, both (a) and (b) raise NoMethodError. But "".foo can be called
from within the refine block.

```
|And, how about the following example, where a nested module is defined?
|
|module R
|  refine String do
|    def foo; p :foo; end
|  end
|
|  module M
|    "".foo
|  end
|  "".foo
|end
|"".foo
```

Every "".foo in the above example should raise NoMethodError, because
they are outside of refine blocks. I admit I've been less careful
about nested refinement modules. For nested refinement modules, it
should behave as following:

```
module R
refine String do
def foo; p :foo; end
end

module M
refine Array do
"".foo  # => OK
end
end
end

using R::M
"".foo  # => NG
```

|> In addition, Module#include should add refinements to included modules, e.g.
|
|This is very different from the current feature, so we need a discussion about it.
|What does "add refinements" mean here?
|There are two aspects about refinement addition. They are defined in modules by Module#refine, and activated in certain scopes by using.
|Does "to add refinements" mean to define (or inherit indirectly) refinements in modules, or to activate refinements in modules, or both of them?

I meant included module will provide refinement of combination of including
module(s) and the module itself.

```
|For example, how should the following code behave?
|
|  module R1
|    refine String do
|     def bar
|       p :bar
|     end
|   end
| end
|
| module R2
|   include R1
|   refine String do
|     def foo
|       p :foo
|     end
|   end
|   "".foo
|   "".bar
| end
```

Since all calls of "".foo and "".bar are outside of refine blocks,
they should raise NoMethodError. But R2 should provide refinement to
add method #bar and #foo to String class.

|Finally, how super in refinements should behave in the new spec?

Refinements should come before normal methods, so super in the normal
method will not see a refined method, and super in the refined method
will see a normal method (or other refined method if refinements are
stacked).

The whole point is separation of defining refinements (for library
developers) and using refinements (for library users).

Any more questions?

```
                                matz.
```

#186 - 11/30/2012 12:41 AM - myronmarston (Myron Marston)

I find refinements to be a very interesting idea, but, as with many others, it concerns me that there aren't rubyspecs behind it, there's still undefined edge cases, the main language maintainers are still discussing how it should work, and the existing implementation has a perf impact.

In the past, I've heard RSpec used as a main example of a library that could really benefit from refinements. As of yesterday, I'm the lead maintainer of RSpec, and I'm doubtful whether or not we'll ever use them. Given the fact that we need to maintain support for 1.8.7, 1.9.2 and 1.9.3 for as long as its common for gem authors to support those versions (since we'd like RSpec to be useable to test all the commonly supported rubies), we won't be in a position to use refinements for many years. In the meantime, we've already started significantly reducing the number of monkey patches RSpec does to Object in the 2.x releases, and I'm hoping to reduce this even more for 3.0.

Really, I think a feature like refinements needs to see some *actual production usage* before being included in the language. It needs a long vetting period. I think a key part of rails' success is the fact that it was extracted from a production app, rather than being built on its own first.

I haven't read this whole thread (it's huge, and it's hard to find the time to read every comment), so I have no idea if this has been discussed or not...but here's an idea I just had: For 2.0, extract refinements out of the core language into a c-extension gem. Ask the community to give refinements a shot. Encourage actual production usage. Learn from real-world usage, and clarify the spec before considering putting them back in the core language.

Of course, this idea depends upon it being doable to extract refinements into a gem; given the affect they have at the core language level, that may not be doable. But it would be great to have some way for refinements to be released so people could start trying them *without* putting them directly in MRI -- that gives the language designers and maintainers time to learn from real world usage and refine refinements (sorry for the bad pun).

Myron

#187 - 11/30/2012 12:42 AM - trans (Thomas Sawyer)

=begin
headius (Charles Nutter) Your idea of having refinements in class hierarchy is very interesting. It is remarkably similar to Cuts. If you are not familiar with cuts see http://rubyworks.github.com/cuts/rcr.html (yea an RCR, remember those!). In fact, I think it means that refinements could be seen as a subset of cuts. In the case of refinements, the methods apply only if the refinement is used in the given scope. This suggests an important piece of functionality from the original cuts concept could use, namely, a condition that applies to the cut's application. If we added that, then refinements could be implement via cuts with:

cut RefinementX < String
def self.applies?(scope, method=nil)
scope.include_refinement? self
end

```
def blah; end
```

end
=end

#188 - 11/30/2012 01:05 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Myron, congrats for the lead maintainer position in RSpec. You're really doing a great job there.

With regards to your suggestion, I think I've read another comment suggesting something like you want but most probably more doable for the time being. Instead of extracting the code to an external gem (which I would guess it won't be done) it would be disabled instead by default (using or refine would raise NoMethodError). Then it could be enabled for experimental usage with either some external c-extension gem that would expose it or through some ruby flag.

I guess this should be much more simpler to implement and would have the same effect you desire, right? I also vote for such approach and delay refinements a bit more until its specs are refined.

#189 - 11/30/2012 01:09 AM - myronmarston (Myron Marston)

rosenfeld (Rodrigo Rosenfeld Rosas) -- yeah, that sounds like it would work fine, as long as there wasn't a perf impact just by having the code there in MRI w/ it turned off.

#190 - 11/30/2012 01:53 AM - drbrain (Eric Hodel)

On Nov 29, 2012, at 4:56, "steveklabnik (Steve Klabnik)" steve@steveklabnik.com wrote:

> Wasn't a 'feature freeze' declared on October 24th? Why are major additions to the language being modified post-freeze, so close to the deadline for an actual release?

Matz decided which features would be in Ruby 2.0.0 on October 24. Between then and the code freeze coming in December we have been completing those accepted features, but not accepting new features.

#191 - 11/30/2012 04:05 AM - headius (Charles Nutter)

Anonymous wrote:

> In message "Re: [ruby-core:50299] [ruby-trunk - Feature #4085] Refinements and nested methods"
> |How should the following code behave?
> |
> |module R
> | refine String do
> | def foo; p :foo; end
> | end
> | "".foo # (a)
> |end
> |"".foo   # (b)
> |
> |Currently, (a) prints :foo, and (b) raises a NoMethodError.
>
> Refinements will be available only from:
>
> - the scope where refinements are added by calling "using"
> - or inside of refine blocks
>
> Inside of refine blocks (not whole module scope) might be
> controversial, but I think it's OK to restrict refinements there. As
> a result, both (a) and (b) raise NoMethodError. But "".foo can be called
> from within the refine block.

I assume this is so that refinement methods will see each other, yes?

module X
refine String do
def foo; :foo; end
def bar; foo; end # should be able to call foo
end
end

So all call sites within a refine block must look for refinements. Now...I would assume they can only see refinements in the same block, but...

> Every "".foo in the above example should raise NoMethodError, because
> they are outside of refine blocks. I admit I've been less careful
> about nested refinement modules. For nested refinement modules, it
> should behave as following:

```
module R
refine String do
def foo; p :foo; end
end

module M
refine Array do
"".foo  # => OK
end
end
end

using R::M
"".foo  # => NG
```

This is incredibly confusing to me. Why are the String refinements active within the refine Array block? That module:

- Is not within the refine String block
- Is not within a file that uses the String refinement
- Does not refine String

Why can it see the String refinements even though it's not in a "using" file and not within a refine String block?

> I meant included module will provide refinement of combination of including
> module(s) and the module itself.

This seems mostly reasonable, since modules that include modules carry all those modules with them to the eventual include site. So using basically considers the entire hierarchy of modules.

```
|For example, how should the following code behave?
|
|  module R1
|    refine String do
|      def bar
|        p :bar
|      end
|    end
|  end
|
|  module R2
|    include R1
|    refine String do
|      def foo
|        p :foo
|      end
|    end
|    "".foo
|    "".bar
|  end
```

> Since all calls of "".foo and "".bar are outside of refine blocks,
> they should raise NoMethodError. But R2 should provide refinement to
> add method #bar and #foo to String class.

More interesting, given the above modules... I assume this would work:

```
#some file that's loaded the above refinements
using R2

"".foo # => OK
"".bar # => OK
```

> |Finally, how super in refinements should behave in the new spec?
>
> Refinements should come before normal methods, so super in the normal
> method will not see a refined method, and super in the refined method
> will see a normal method (or other refined method if refinements are
> stacked).

By stacking, I assume you mean something like this:

```
class String
def bar; p :base; end
end

module M1
refine String do
def bar; p :m1; super end
end
end

module M2
refine String do
def bar; p :m2; super; end
end
end

using M1
using M2

"".bar # => should print :m2, :m1, :base
```

Correct?

My idea of *actually* putting refinements into the hierarchy seems like it might fit this best, rather than searching two hierarchies. Will have to think about it more.

> Any more questions?

Yes, lots! :)

- Will we make more methods have special access to the caller's frame to support refinements? I'm speaking here of modifications to send, method reflection, to_proc, and so on. Of these, send sorta has access to caller frame, but it's unspecified (send :binding behaves differently on different impls). The reflection methods (method, instance_method, ...) and to_proc do not currently have access to caller's frame, and would have to be special-cased (and in JRuby that would force deoptimization of code that calls them to ensure we have a frame available). I would argue that *none* of these methods should see the effects of refinements, because they are not defined within a refined file or scope. Making them see refinements extends refined effects down-stack in unpredictable ways.

- Questions about about the scoping of refinements in seemingly unrelated refine blocks.

- Performance expectations. Is performance being completely ignored here, or shall we set some goals for the implementation? I appreciate your desire to make Ruby more expressive, but that doesn't pay the bills at the end of the day if refined calls are 100x slower than regular calls. Some consideration of performance needs to be made up front. I would propose that unless refined calls can be made exactly the same speed as unrefined calls, they should not be included. We'd essentially be adding a file-scoped feature that hurts performance of all calls in that file. Expressivity doesn't trump slowing an entire system down because of a single call made at the top level of a file.

I'm glad to see refinements being reduced in scope, but we still have a lot to talk through. I share others' concerns about this feature not being settled just a couple months before 2.0 is supposed to be released, and at this point I'd still rather see it delayed to 2.1 so we can work through all the edges. I do not yet have a clear picture in my head of how refinements are supposed to be structured, and will have to reboot the understanding I did have based on recent updates.

I will try to modify my refinements implementation to work according to the new specification and see how it feels, implementation-wise.

FWIW, I made an improvement to JRuby 1.7.2 that makes the calling scope (the static part where cref and refinements live) available to all methods, even when scope/frame have been optimized away. It may make it possible to implement refinements without reducing the number of optimizations we do.

To others on this thread: I'd really like to hear more substantive input on the feature, rather than just "please don't do this, it's bad, I don't like it, it's too late, wah wah." If you have a specific aspect you don't like, discuss that. If you don't like the feature in toto, explain why. Special pleading for it to be removed does not further the conversation.

#192 - 11/30/2012 04:35 AM - Gibheer (Stefan Radomski)

As headius wants substantive questions, here are some

- What is the plan to make sure, that this feature is compatible to future versions? You introduce a behavior now which depends on 'using' being at the beginning of every file, which wants to use it. If the behavior should be changed in the direction of the original proposal, will that still be compatible?

- As I read matz example (#175), I can't find any difference in the code. Why exactly is it not working the second time and why only #bar?

#193 - 11/30/2012 06:23 AM - Anonymous

In message "Re: [ruby-core:50338] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Fri, 30 Nov 2012 04:05:15 +0900, "headius (Charles Nutter)" headius@headius.com writes:

|I assume this is so that refinement methods will see each other, yes?

Yes.

```
|module X
|  refine String do
|    def foo; :foo; end
|    def bar; foo; end # should be able to call foo
|  end
|end
|
```

|So all call sites within a refine block must look for refinements. Now...I would assume they can only see refinements in the same block, but...

No, whole refinement provided by module X can be seen from refine
block in module X.

|This is incredibly confusing to me. Why are the String refinements active within the refine Array block? That module:
|
|* Is not within the refine String block
|* Is not within a file that uses the String refinement
|* Does not refine String
|
|Why can it see the String refinements even though it's not in a "using" file and not within a refine String block?

Because it's within the body of refinement module X. Since both
refinements are defined in the same refinement, we consider they are
related (you can't see the relation from this silly examples).

|> Refinements should come before normal methods, so super in the normal
|> method will not see a refined method, and super in the refined method
|> will see a normal method (or other refined method if refinements are
|> stacked).
|
|By stacking, I assume you mean something like this:

```
|
|class String
|  def bar; p :base; end
|end
|
|module M1
|  refine String do
|    def bar; p :m1; super end
|  end
|end
|
|module M2
|  refine String do
|    def bar; p :m2; super; end
|  end
|end
|
|using M1
|using M2
|
|"".bar # => should print :m2, :m1, :base
|
```

|Correct?

Correct.

|My idea of *actually* putting refinements into the hierarchy seems like it might fit this best, rather than searching two hierarchies. Will have to think about it more.

It makes refinement decoration quite unpredictable. For example,

```
module M
refine Integer do
def /(n); self / n.to_f; end
end
end
```

using M
1 / 2 # => you expect 0.5

But if refinements are searched through inheritaance hierarchy, it
won't work since there's Fixnum#/.

|> Any more questions?
|
|Yes, lots! :)

I will answer each of them later.

```
                              matz.
```

#194 - 11/30/2012 10:00 AM - shugo (Shugo Maeda)

Thanks for your answering my questions. I understand your intention.

matz wrote:

> In message "Re: [ruby-core:50299] [ruby-trunk - Feature #4085] Refinements and nested methods"
> on Thu, 29 Nov 2012 15:02:03 +0900, "shugo (Shugo Maeda)" redmine@ruby-lang.org writes:
>
> |> * refinements are file scope
> |> * only top-level "using" is available
> |> * no module scope refinement
> |
> |Do these constraints just mean that main.using is available, but Module#using is not?
>
> Yes, only main.using should be available. No Module#using (for 2.0).

So, the current behavior of main.using need not be changed, right?
Technically, the current behavior of main.using is not file scope.
The scope of main.using is from the point where using is *called at runtime* to the end of that file.
For example,

p 1 / 2 # => 0 (not refined)
using MathN # MathN refines Fixnum#/
p 1 / 2 # => (1/2) (refined)

And a more complex example is:

if false
using MathN
end
p 1 / 2 #=> 0 (not refined)

> |How should the following code behave?
> |
> |module R
> | refine String do
> | def foo; p :foo; end
> | end
> | "".foo # (a)
> |end
> |"".foo # (b)
> |
> |Currently, (a) prints :foo, and (b) raises a NoMethodError.
>
> Refinements will be available only from:
>
> - the scope where refinements are added by calling "using"
> - or inside of refine blocks
>
> Inside of refine blocks (not whole module scope) might be
> controversial, but I think it's OK to restrict refinements there. As
> a result, both (a) and (b) raise NoMethodError. But "".foo can be called
> from within the refine block.

"Inside of refine blocks" means that all refinements defined in a module are activated
in all refine blocks in that module and nested modules, right?

There are some considerations:

- It may be better not to support nested module to simplify things. I think support for nested modules are important for refinement users, but are not so important for refinement authors.

- It may be hard to implement efficiently activation of refinements which are defined after refine blocks.
  For example,
  module M
  refine Array do
  def to_json; "[" + map { |i| i.to_json } + "]" end
  end

  refine Hash do
  def to_json; "{" + map { |k, v| k.to_s.dump + ":" + v.to_json } + "}" end
  end
  end
  It may be better to limit refinement activation in refine blocks to refinements to be defined by the refine blocks theselves, to simplify things. If there's no refinement activation in refine blocks, recursive methods cannot be defined, so the refinement to be defined should be activated at least.

- Refinement activation in refine blocks may have the same problem as refinement-aware module_eval,
  for example in the following code:

  F = Proc.new { 1 / 2 }
  module M
  refine Fixnum do def /(other) quo(other) end end
  refine(Object, &F)
  end

  |There are two aspects about refinement addition. They are defined in modules by Module#refine, and activated in certain scopes by using.
  |Does "to add refinements" mean to define (or inherit indirectly) refinements in modules, or to activate refinements in modules, or both of them?

  I meant included module will provide refinement of combination of including
  module(s) and the module itself.

It does make sense.
What happens if the same class is refined both in a module and another module included into that module?
For example,

class C
def foo; p :C; end
end
module M1
refine String do def foo; p :M1; super; end; end
end
module M2
include M1
refine String do def foo; p :M2; super; end; end
end
using M2
C.new.foo #=> ?

I think it's better to just calls M2 and C, not M1, to simplify things.
super chain is too complex here.

> |Finally, how super in refinements should behave in the new spec?
>
> Refinements should come before normal methods, so super in the normal
> method will not see a refined method, and super in the refined method
> will see a normal method (or other refined method if refinements are
> stacked).

Charles showed an edge case.

module A; refine(Numeric) { def blah; puts 'A'; super; end }; end
module B; refine(Integer) { def blah; puts 'B'; super; end }; end
module C; refine(Fixnum) { def blah; puts 'C'; super; end }; end
using A; using B; using C
1.blah

Currently, only C is called, but what should happen?
At first, I thought all blah should be called, but super in C is in scope of neither A nor B,
it might be better not to call A and B.

I'm starting to think it might be better to limit super to call only the original method in the refined class to simplify the spec.

For example,

class X; def blah; puts 'X'; end
module A; refine(X) { def blah; puts 'A'; super; end }; end
module B; refine(X) { def blah; puts 'B'; super; end }; end
module C; refine(X) { def blah; puts 'C'; super; end }; end
using A; using B; using C
1.blah

Only C and X is called in the above code.
At first, I thought that stacking refinements and super chain are useful for aspect oriented programming.
But refinements have no local rebinding, so it might not be a real use case of refinements.

#195 - 11/30/2012 10:23 AM - Anonymous

In message "Re: [ruby-core:50338] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Fri, 30 Nov 2012 04:05:15 +0900, "headius (Charles Nutter)" headius@headius.com writes:

|> Any more questions?
|
|Yes, lots! :)
|
|* Will we make more methods have special access to the caller's frame to support refinements? I'm speaking here of modifications to send, method reflection, to_proc, and so on. Of these, send sorta has access to caller frame, but it's unspecified (send :binding behaves differently on different impls). The reflection methods (method, instance_method, ...) and to_proc do not currently have access to caller's frame, and would have to be special-cased (and in JRuby that would force deoptimization of code that calls them to ensure we have a frame available). I would argue that *none* of these methods should see the effects of refinements, because they are not defined within a refined file or scope. Making them see refinements extends refined effects down-stack in unpredictable ways.

From usability perspective, all retrospective methods e.g. respond_to?
methods etc. should reflect refinements. But for 2.0, I don't make
them check refinements, because of performance and complexity. It
should be issue of future version.

|* Questions about about the scoping of refinements in seemingly unrelated refine blocks.
|
|* Performance expectations. Is performance being completely ignored here, or shall we set some goals for the implementation? I appreciate your desire to make Ruby more expressive, but that doesn't pay the bills at the end of the day if refined calls are 100x slower than regular calls. Some consideration of performance needs to be made up front. I would propose that unless refined calls can be made exactly the same speed as unrefined calls, they should not be included. We'd essentially be adding a file-scoped feature that hurts performance of all calls in that file. Expressivity doesn't trump slowing an entire system down because of a single call made at the top level of a file.

My expectation is that if you don't see "using" and "refine" in the
source code, the performance will not be affected, as much as
possible. It's the reason I gave up refinements in the module bodies,
inherited modules/classes and module_eval. I don't demamd refined
call to be exact same speed as unrefined ones, but I'd expect them not
significantly slower. I'd accept them if they are 5-15% slower than
normal calls.

```
                                matz.
```

#196 - 11/30/2012 10:42 AM - trans (Thomas Sawyer)

=begin
matz (Yukihiro Matsumoto)

> It makes refinement decoration quite unpredictable. For example,
>
> module M
> refine Integer do
> def /(n); self / n.to_f; end
> end
> end
> using M
> 1 / 2 # => you expect 0.5
>
> But if refinements are searched through inheritaance hierarchy, it
> won't work since there's Fixnum#/.

Then I'd say they refined the wrong class. They should have refined Fixnum. If refining Integer somehow places the refinement in front of Fixnum, then I think all sorts of craziness might ensue. Consider:

```
class A
def x(i); i; end
end

class B < A
def x(i); super ** 2; end
end

A.new.x(3)  #=> 3
B.new.x(3)  #=> 9

...

module Moo
refine A do
def x(i); super + 1; end
end
end

using Moo

A.new.x(3)  #=> 4
B.new.x(3)  #=> 10  # not 16!?

=end
```

#197 - 11/30/2012 10:58 AM - trans (Thomas Sawyer)

> |What happens if the same class is refined both in a module and another module included into that module?
> |For example,
> |
> | class C
> | def foo; p :C; end
> | end
> | module M1
> | refine String do def foo; p :M1; super; end; end
> | end
> | module M2
> | include M1
> | refine String do def foo; p :M2; super; end; end
> | end
> | using M2
> | C.new.foo #=> ?
> |
> |I think it's better to just calls M2 and C, not M1, to simplify things.
> |super chain is too complex here.
>
> I was thinking of M2->M1->C, but M2->C is simpler and acceptable.

This can't be b/c then you can't refine a previous refinement. Thus it breaks modularity (black box) principle. E.g. if I require 'x.rb' and apply using B, it should not matter if x.rb is using A or not. (I can explain that with a detail example if it is not clear enough).

#198 - 11/30/2012 11:04 AM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

> By stacking, I assume you mean something like this:
>
> ```
> class String
> def bar; p :base; end
> end
>
> module M1
> refine String do
> def bar; p :m1; super end
> end
> end
>
> module M2
> refine String do
> def bar; p :m2; super; end
> end
> ```

> end
>
> using M1
> using M2
>
> "".bar # => should print :m2, :m1, :base
>
> Correct?
>
> My idea of *actually* putting refinements into the hierarchy seems like it might fit this best, rather than searching two hierarchies. Will have to think about it more.

I guess your idea is similar to ko1's idea using prepend to implement refinements.
What happens when refinements are used in a different order in a different file?

a.rb:
using M1
using M2
"".bar # => should print :m2, :m1, :base

b.rb:
using M2
using M1
"".bar # => what happens?

I think it might be better to abandon stacking refinements as I said in another comment.

#199 - 11/30/2012 11:23 AM - Anonymous

Hi,

In message "Re: [ruby-core:50355] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Fri, 30 Nov 2012 10:43:04 +0900, "trans (Thomas Sawyer)" transfire@gmail.com writes:

|Then I'd say they refined the wrong class. They should have refined Fixnum. If refining Integer somehow places the refinement in front of Fixnum, then I think all sorts of craziness might ensue.

Otherwise the refinement will be more fragile. Fixnum is
implementation detail. For example:

class Foo
end
class FooImpl < Foo
end
class FooImpl2 < Foo
end

FooImpl and FooImpl2 are implementation detail

module X
refine Foo do
def x; ...; end
end
end

we want to intercept method x of class X (and its subclasses).
we don't want to step in to implementation detail, if possible.

```
|  # foo.rb library
|  class A
|    def x(i); i; end
|  end
|  class B < A
|    def x(i); super ** 2; end
|  end
|
|  A.new.x(3)  #=> 3
|  B.new.x(3)  #=> 9
|
|  # bar.rb
|  require 'foo'
|

|  module Moo
|   refine A do
|     def x(i); super + 1; end
|   end
| end
|
| using Moo
|
| A.new.x(3)  #=> 4
| B.new.x(3)  #=> 10  # not 16!?
```

Some may expect 10, and others may expect 16. We cannot satisfy them
all at once. It's matter of design choice.

```
                            matz.
```

#200 - 11/30/2012 11:23 AM - Anonymous

In message "Re: [ruby-core:50357] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Fri, 30 Nov 2012 11:02:10 +0900, "trans (Thomas Sawyer)" transfire@gmail.com writes:

|> I was thinking of M2->M1->C, but M2->C is simpler and acceptable.
|
|This can't be b/c then you can't refine a previous refinement. Thus it breaks modularity (black box) principle. E.g. if I require 'x.rb' and apply using B, it should not matter if x.rb is using A or not. (I can explain that with a detail example if it is not clear enough).

You are right, by this spec you can refine a method only once in a
file. But I don't understand how it breaks black box principle.
it does not change the behavior if required x.rb is 'using A' or not,
under the current spec.

```
                            matz.
```

#201 - 11/30/2012 12:23 PM - shugo (Shugo Maeda)

Your mail seems to be failed to synced into bugs.ruby-lang.org, so I
reply by e-mail.

2012/11/30 Yukihiro Matsumoto matz@ruby.or.jp:

> |So, the current behavior of main.using need not be changed, right?
> |Technically, the current behavior of main.using is not file scope.
> |The scope of main.using is from the point where using is *called at runtime* to the end of that file.
> |For example,
> (snip)
> In that sense, I don't think we need to change the behavior. What I
> meant by "file scope" was that refinement only available until end of
> the file.

I see.

> |* It may be better not to support nested module to simplify things.
> | I think support for nested modules are important for refinement users, but are not so important
> | for refinement authors.
>
> I don't think nested modules are important. We can drop them, at
> least for 2.0 to simplify things.

OK.

```
|* It may be hard to implement efficiently activation of refinements which are defined after refine blocks.
|  For example,
|    module M
|      refine Array do
|        def to_json; "[" + map { |i| i.to_json } + "]" end
|      end
|
|      refine Hash do
|        def to_json; "{" + map { |k, v| k.to_s.dump + ":" + v.to_json } + "}" end
|      end
|    end
|  It may be better to limit refinement activation in refine blocks to refinements to be defined by the refine blocks theselves, to simplify things.  If there's no refinement activation in refine blocks, recursive methods cannot be defined, so the refinement to be defined should be activated at least.
```

I am not sure if I understand you correctly. I thought method look-up
should be done in run-time. So only I can say it that refinement M
will be available in both refine blocks in the above example.

In the current implementation the activated refinement table of a
module is shared by the code in that module definition
including refine blocks using cref.
However, to limit refinement activation only in refine blocks, cref
can't be used for that purpose, so a little hack is needed.
I think it's possible, at least in CRuby, to store hidden table in a
module, and share that table in refine blocks.

```
|* Refinement activation in refine blocks may have the same problem as refinement-aware module_eval,
|  for example in the following code:
|
|    F = Proc.new { 1 / 2 }
|    module M
|      refine Fixnum do def /(other) quo(other) end end
|      refine(Object, &F)
|    end
```

Yes, but I consider its behavior implementation dependent.
Fundamentally you cannot expect passing proc to lambda work correctly.

I see.

```
|  class C
|    def foo; p :C; end
|  end
|  module M1
|    refine String do def foo; p :M1; super; end; end
|  end
|  module M2
|    include M1
|    refine String do def foo; p :M2; super; end; end
|  end
|  using M2
|  C.new.foo #=> ?
|
|I think it's better to just calls M2 and C, not M1, to simplify things.
|super chain is too complex here.
```

I was thinking of M2->M1->C, but M2->C is simpler and acceptable.

M2->M1->C may be possible, but I worry that it makes super chain more complex.

```
|Charles shown an edge case.
|
|module A; refine(Numeric) { def blah; puts 'A'; super; end }; end
|module B; refine(Integer) { def blah; puts 'B'; super; end }; end
|module C; refine(Fixnum) { def blah; puts 'C'; super; end }; end
|using A; using B; using C
|1.blah
|
|Currently, only C is called, but what should happen?
|At first, I thought all blah should be called, but super in C is in scope of neither A nor B,
|it might be better not to call A and B.
|
|I'm starting to think it might be better to limit super to call only the original method in the refined class to simplify the spec.
```

So do you mean refined method appear only once in method look-up chain?

I mean that all refinements appear in the method lookup 1.blah, but
the only first found one is used, and succeeding super in C outside
the scope of using are not affected by other refinements A and B.
make sense?

```
|For example,
|
|class X; def blah; puts 'X'; end
|module A; refine(X) { def blah; puts 'A'; super; end }; end
```

|module B; refine(X) { def blah; puts 'B'; super; end }; end
|module C; refine(X) { def blah; puts 'C'; super; end }; end
|using A; using B; using C
|1.blah

X.new.blah # <= do you mean X here?

Yes.

|Only C and X is called in the above code.
|At first, I thought that stacking refinements and super chain are useful for aspect oriented programming.
|But refinements have no local rebinding, so it might not be a real use case of refinements.

Fair enough. If we can warn users for conflicted refinements like
above, it's even better.

I'd like to hear other opinions, especially Charles' one.

--
Shugo Maeda

#202 - 11/30/2012 02:41 PM - jonforums (Jon Forums)

|* Performance expectations. Is performance being completely ignored here, or shall we set some goals for the implementation? I appreciate your desire to make Ruby more expressive, but that doesn't pay the bills at the end of the day if refined calls are 100x slower than regular calls. Some consideration of performance needs to be made up front. I would propose that unless refined calls can be made exactly the same speed as unrefined calls, they should not be included. We'd essentially be adding a file-scoped feature that hurts performance of all calls in that file. Expressivity doesn't trump slowing an entire system down because of a single call made at the top level of a file.

My expectation is that if you don't see "using" and "refine" in the
source code, the performance will not be affected, as much as
possible. It's the reason I gave up refinements in the module bodies,
inherited modules/classes and module_eval. I don't demamd refined
call to be exact same speed as unrefined ones, but I'd expect them not
significantly slower. I'd accept them if they are 5-15% slower than
normal calls.

Does the current implementation yet allow you guys to ballpark the range of performance impact? Specifically, the multiple gem (activated) scenario in which each gem has been refined.

If yes, does the performance impact appear to be pay-once-all-you-can-eat or does the decrease appear to be a function (linear, log, exp?) of the number of active gems in your graph?

Jon

#203 - 11/30/2012 08:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 29-11-2012 23:42, Yukihiro Matsumoto escreveu:

```
|  class C
|    def foo; p :C; end
|  end
|  module M1
|    refine String do def foo; p :M1; super; end; end
|  end
|  module M2
|    include M1
|    refine String do def foo; p :M2; super; end; end
|  end
|  using M2
|  C.new.foo #=>  ?
|
```

|I think it's better to just calls M2 and C, not M1, to simplify things.
|super chain is too complex here.

I was thinking of M2->M1->C, but M2->C is simpler and acceptable.

I'm worried about this. We shouldn't be defining this behavior in light
of what is simple/feasible until 2.0 release. Because if this decision

changes later it will be backward incompatible and it is most likely
that we won't be able to change this behavior in the future to keep
compatibility.

It would be better to think well about this now. What is the really
desired behavior in the long-run to avoid compatibility issues in the
future?

#204 - 12/01/2012 03:59 AM - The8472 (Aaron G)

> Since there still remain undefined corner case behavior in refinements, and the time is running out, I decided not to introduce full refinement for Ruby 2.0. The limited Ruby 2.0 refinement spec will be:
>
> - refinements are file scope
> - only top-level "using" is available
> - no module scope refinement
> - no refinement inheritance
> - module_eval do not introduce refinement (even for string args)

While this removes a number of potential causes of problems we're now
stuck with a feature-set that can not cover many of the originally
discussed use-cases.

And to support the restricted feature set we already have disagreements
where refinements should be injected in the method hierarchy. What is
labled as a "design decision" aren't even edge cases anymore. They're
essential behavior of a new feature decided without much discussion or
analysis in the context of potential uses.

For DSLs we already have BasicObject + method_missing + instance_eval,
so only ruby literals may need some sugar added to make the DSL really
fluent. But if you already have all the instance_eval magic going then
you don't really want to require people to pepper their code with
"using" just so they can use your magical DSL.

For patching broken/incompatible code Module.prepend is much better
anyway since the breakage may occur in 3rd party code where you can't
use "using" anyway.

So with what does this leave us? String.camelize and 2.days.ago? Those
are supposed to be convenience methods.

If your files end up with a header of

```
using "EnumeratorExtensions", "ActiveSupport::ClassNameConversions",
```

"MyApplication::DateHelpers", "SomeGem::HashExtensions",
"MyApplication::ActiveRecordExtensions", ...

That's not convenient at all anymore.

Personally I wouldn't want to use refinements as they are now. I would
love to have a powerful tool like pointcuts or Module.prepend in my
toolbox. Refinements wouldn't really qualify.

I'm getting the impression that refinements are getting rushed despite
being supposed to bring a completely new axis to the language along
which we can compose our applications. This shouldn't be happening in a
mature language. The whole way this is going right now feels wrong to me.

So I'm asking for two things: a) please consider removing refinements
from ruby 2.0. b) rethink how such big features should be designed in
the future.

#205 - 12/01/2012 04:41 AM - headius (Charles Nutter)

Anonymous wrote:

> From usability perspective, all retrospective methods e.g. respond_to?
> methods etc. should reflect refinements. But for 2.0, I don't make
> them check refinements, because of performance and complexity. It
> should be issue of future version.

respond_to? brings up a really good point: there's lots of methods that might need special refinement care...but I think it makes things more confusing rather than less.

Let's take the respond_to? example. You want respond_to? to reflect refinements. That seems reasonable on the surface, even though respond_to? is not refined itself. The first peculiarity there is that you could no longer wrap respond_to?, respond_to_missing?, method, instance_method, and so on, because wrapping them would break their visibility to refinements (due to the intervening unrefined Ruby frame). We'd be reducing the metaprogrammability of many, many core methods.

And then there's methods that *call* respond_to? (or coercion methods like to_s). Example:

class X; end

module M1
refine X
def to_path; '/tmp'; end
end
end

using M1

File.open(X.new) # ????

File.open checks respond_to?(:to_path) and if that succeeds it calls to_path. But the above code will never work unless File.open is also made refinement-aware.

I think this is a pretty significant problem, and it shows how much more limited refinements will actually be. The bottom line is that making any core methods reflect refinements may be *more* confusing, because only direct invocations will work...not wrapped invocations, called-method invocations, or double-dispatched invocations.

Again it may be good to look at C# extension methods, which are not reflected:
http://stackoverflow.com/questions/299515/c-sharp-reflection-to-identify-extension-methods

More and more I see the only value of refinements as providing extension methods for specific classes in specific scopes. If we want reflective access, we should provide something like C# that allows getting the available refined methods and a mechanism that returns the current active refinements. Users can figure out what they need from there without special-casing a whole bunch of core methods.

class Module
def refined_methods ...
def refined_instance_methods ...
end

module Kernel
def active_refinements ...
end

More coming...

#206 - 12/01/2012 04:54 AM - headius (Charles Nutter)

Anonymous wrote:

> |This is incredibly confusing to me. Why are the String refinements active within the refine Array block? That module:
> |
> |* Is not within the refine String block
> |* Is not within a file that uses the String refinement
> |* Does not refine String
> |
> |Why can it see the String refinements even though it's not in a "using" file and not within a refine String block?
>
> Because it's within the body of refinement module X. Since both
> refinements are defined in the same refinement, we consider they are
> related (you can't see the relation from this silly examples).

I'm still very confused by this. You said this won't work:
module R
refine String do
def foo; p :foo; end
end

module M
"".foo
end
"".foo

end
"".foo

Shugo posted this example and you replied "Every "".foo in the above example should raise NoMethodError, because they are outside of refine blocks."

However, you go on to say that the example below should work, and I don't understand why:

module R
refine String do
def foo; p :foo; end
end

module M
refine(Array) { ... }
"".foo
end
"".foo
end
"".foo

Nothing has changed here; none of the foo calls are within refine blocks, but your example indicates the innermost foo should work. Why?

> |My idea of *actually* putting refinements into the hierarchy seems like it might fit this best, rather than searching two hierarchies. Will have to think about it more.
>
> It makes refinement decoration quite unpredictable. For example,
>
> module M
> refine Integer do
> def /(n); self / n.to_f; end
> end
> end
> using M
> 1 / 2 # => you expect 0.5
>
> But if refinements are searched through inheritaance hierarchy, it
> won't work since there's Fixnum#/.

This shouldn't work anyway. Why would we look at refinements for Integer if Fixnum has overridden those methods? This seems completely anti-OO to me. Are you saying refinements can route completely around overridden methods without touching the class that overrides them?

Refinements should apply to a given type, visible to subtypes only if method searching leads it there. We're going to have mass confusion if overridden methods on a subclass can be simply brushed away by refined methods on the superclass. And I'm not even sure how to implement that...we'd have to do a full hierarchy search of all classes looking for refined methods *before* we do a normal search of those same classes. And in your Integer case, what woule super do? Should it call Fixnum#/ or Integer#/? I don't think either is right, and I think being able to route around the overridden methods in a child class is dead wrong.

#207 - 12/01/2012 05:02 AM - headius (Charles Nutter)

Anonymous wrote:

> Hi,
>
> In message "Re: [ruby-core:50355] [ruby-trunk - Feature #4085] Refinements and nested methods"
> on Fri, 30 Nov 2012 10:43:04 +0900, "trans (Thomas Sawyer)" transfire@gmail.com writes:
>
> |Then I'd say they refined the wrong class. They should have refined Fixnum. If refining Integer somehow places the refinement in front of Fixnum, then I think all sorts of craziness might ensue.
>
> Otherwise the refinement will be more fragile. Fixnum is
> implementation detail. For example:
>
> class Foo
> end
> class FooImpl < Foo
> end
> class FooImpl2 < Foo
> end
>
> # FooImpl and FooImpl2 are implementation detail
>
> module X
> refine Foo do

> def x; ...; end
> end
> end
>
> # we want to intercept method x of class X (and its subclasses).
> # we don't want to step in to implementation detail, if possible.

I believe you are incorrect. FooImpl1 and FooImpl2 are not simply implementation details...they're critical parts of the OO hierarchy.

Refinements are supposed to localize monkey-patching, but what you want here is way, way beyond monkey-patching, It's not possible to monkey-patch Foo to have an x method if either FooImpl1 or FooImpl2 define their own. Refinements should not be able to route around overridden methods, or the entire structure of an OO hierarchy becomes meaningless.

```
| # foo.rb library
| class A
|   def x(i); i; end
| end
| class B < A
|   def x(i); super ** 2; end
| end
|
| A.new.x(3)  #=> 3
| B.new.x(3)  #=> 9
|
| # bar.rb
| require 'foo'
|

| module Moo
|   refine A do
|     def x(i); super + 1; end
|   end
| end
|
| using Moo
|
| A.new.x(3)  #=> 4
| B.new.x(3)  #=> 10  # not 16!?
```

> Some may expect 10, and others may expect 16. We cannot satisfy them
> all at once. It's matter of design choice.

Nobody should expect 10...if they do they're simply wrong. B defines its own x method which has been neither monkeypatched nor refined. B's original x should be called.

Refinements should be like extension methods; available for direct calls within some scope (and subscopes), not reflected down the call stack (no method, send, respond_to? tricks), and they should still honor basic OO structure (no routing around overridden methods because a parent is refined).

#208 - 12/01/2012 05:07 AM - headius (Charles Nutter)

shugo (Shugo Maeda) wrote:

> I guess your idea is similar to ko1's idea using prepend to implement refinements.
> What happens when refinements are used in a different order in a different file?
>
> a.rb:
> using M1
> using M2
> "".bar # => should print :m2, :m1, :base
>
> b.rb:
> using M2
> using M1
> "".bar # => what happens?
>
> I think it might be better to abandon stacking refinements as I said in another comment.

Yes, this doesn't work unfortunately. The super chain is a source of endless problems. Again, C# extension methods do not honor "super" in any way; the methods are defined statically and simply inserted at the point of call. Handling super obviously leads to all sorts of terrible edge cases.

Again I'm going to urge STRONGLY that refinements be moved completely out of 2.0 for possible inclusion in 2.1. I feel like the refinements specification is changing *by the hour* and we do not have enough time to work out all the issues.

#209 - 12/01/2012 05:16 AM - headius (Charles Nutter)

shugo (Shugo Maeda) wrote:

> I mean that all refinements appear in the method lookup 1.blah, but
> the only first found one is used, and succeeding super in C outside
> the scope of using are not affected by other refinements A and B.
> make sense?

This is exactly right.

> |For example,
> |
> |class X; def blah; puts 'X'; end
> |module A; refine(X) { def blah; puts 'A'; super; end }; end

A refines X, and no other refinements are active. super calls X#blah.

> |module B; refine(X) { def blah; puts 'B'; super; end }; end

B refines X, and no other refinements are active. super calls X#blah.

> |module C; refine(X) { def blah; puts 'C'; super; end }; end

C refines X, and no other refinements are active. super calls X#blah.

> |using A; using B; using C
> |1.blah
>
> X.new.blah # <= do you mean X here?
>
> Yes.
>
> |Only C and X is called in the above code.
> |At first, I thought that stacking refinements and super chain are useful for aspect oriented programming.
> |But refinements have no local rebinding, so it might not be a real use case of refinements.
>
> Fair enough. If we can warn users for conflicted refinements like
> above, it's even better.
>
> I'd like to hear other opinions, especially Charles' one.

I think you hit the nail on the head. The important concept here is the call site. Each of the super call sites above appears within the context of a *single* refinement, so they should not dispatch to anything but the original method.

This means super chaining should only work in contexts that *nest* refinements, which I think would be exceedingly rare and not really useful (and maybe should simply be an error):

```
class X; def blah; p 'X'; end; end
module A
refine X do
def blah; p 'A'; super; end
module B
refine X do
def blah; p 'B'; super; end
module C
refine X do
def blah; p 'C'; super; end
end
end
end
end
end
```

end

using A::B::C
X.new.blah => C, B, A, X

Or I suppose it could work in the case where A, B, C are included in order into another module, which is then using'ed, since in that case the refinements are supposed to stack?

#210 - 12/01/2012 06:46 AM - enebo (Thomas Enebo)

Could someone make a summary of the current proposed refinements feature in the next couple of days once the churn of corner cases slows down a bit? I have been reading through this and I have three observations:

1. It takes a long time to read through the thread now and the feature itself changes the lower you go down this issue.
2. Because the feature is changing I see people perhaps making comments which may or may not apply anymore to the reduced-scope version of refinements.
3. This much mutation makes me worry this revised feature won't have enough intellectual bake time to make it into a February/March release.

I would love to see this restated with the corner cases covered based on Matz's last reduced scope statement. Especially in regards to special methods (meta-programming, respond_to?...) and super in particular.

Could a wiki with an official editor (like shugo) help? Then all new users could see the entirety of the current feature without needing to read what is becoming a massive thread.

#211 - 12/01/2012 09:23 PM - The8472 (Aaron G)

On 30.11.2012 20:41, headius (Charles Nutter) wrote:

> respond_to? brings up a really good point: there's lots of methods that might need special refinement care...but I think it makes things more confusing rather than less.
>
> Let's take the respond_to? example. You want respond_to? to reflect refinements. That seems reasonable on the surface, even though respond_to? is not refined itself. The first peculiarity there is that you could no longer wrap respond_to?, respond_to_missing?, method, instance_method, and so on, because wrapping them would break their visibility to refinements (due to the intervening unrefined Ruby frame). We'd be reducing the metaprogrammability of many, many core methods.
>
> And then there's methods that *call* respond_to? (or coercion methods like to_s). Example:
>
> class X; end
>
> module M1
> refine X
> def to_path; '/tmp'; end
> end
> end
>
> using M1
>
> File.open(X.new) # ????
>
> File.open checks respond_to?(:to_path) and if that succeeds it calls to_path. But the above code will never work unless File.open is also made refinement-aware.
>
> I think this is a pretty significant problem, and it shows how much more limited refinements will actually be. The bottom line is that making any core methods reflect refinements may be *more* confusing, because only direct invocations will work...not wrapped invocations, called-method invocations, or double-dispatched invocations.

The common problem of all these methods is that they happen down-stack.
The obvious solution that problem would be the option to make
refinements optionally apply on a thread-scope too, but the performance
implications obviously are horrible as they would could impact any
callsite anywhere in the whole application.

So I think the proper solution is to provide a much much more low-level
metaprogramming alternative: The modification of individual, selected
callsites and methods. I.e. Allow metaprogramming on the AST itself.

Once we can do that we can also inspect the stack - or as potentially
more performant alternative as it wouldn't need to reify the whole stack

- examine thread-local variables.

Modifying the AST obviates the problem of inheritance as the programmer
has precise control of which method or callsite gets modified where.

To express some commonly thrown-around refinement examples as
AST-transforms:

(Concepts shamelessly stolen from Java's MethodHandle/invokedynamic and
AspectJ)

case a) Traditional monkeypatching

```
class A
  def foo
    puts "in A"
    raise "woops"
  end
end
```

```
class B < A
  def foo
    super
    puts "after exception!"
  end
end
```

```
Ruby::AST.methods(A, :foo).enter do |method_name, *args|
  puts "inserted in #{self.name}" # we're inside the method here
end
```

```
A.new.foo
# inserted in A
# in A
# Exception: woops
```

```
# match super call
sites = Ruby::Ast.methods(B, :foo).callsites(A, :foo)
# match existing AST callsites and those defined in the future
transform = sites.transforms(:existing, :future)
```

```
# add a new transform
sites.wrap do |source_self,target_self,target_as_proc, *args, &block|
  nil # don't execute target method
end
```

```
B.new.foo
# after exception!
```

case b) Simple, scoped Intercept

```
class C
  def m1
    "Foo".downcase
  end
end
```

```
# match methods in C, then select callsites inside those methods
sites = Ruby::AST.methods(C, [:m1,:m2]).callsites(String, :downcase)
```

```
# don't modify existing code
sites.transforms(:future).after do |target, result, *args|
  result + "x"
end
```

```
class C
  def m2
    "Foo".downcase
  end

  def m3
    "Foo".downcase
  end
end
```

```
C.new.m1 # => foo
C.new.m2 # => foox
C.new.m3 # => foo
```

case c) advanced example, down-stack "refinement"

```
f = future_transforms = []
```

```
# not scoped to specific methods here!
sites = Ruby::AST.callsites(String, :dasherize)
transform = sites.transforms(:future, :existing)
transform = transform.guard{Thread.token?(:refine_dasherize)}
```

```
# modifies callsites in the whole application
# megamorphic ones will suffer a performance loss from a typecheck.
# optimized, monomorphic callsites will only
#  suffer from the thread-local variable check
# since it's not volatile it can be hoisted in loops
f << transform.wrap do |target,target_method_as_proc, *args, &block|
   if target_method_as_proc != nil
      # ok, target already exists, let's call that
      target_method_as_proc.call(*args, &block)
   else
      # roll our own implementation
      target.gsub(%r_/, '-')
   end
end
```

```
sites = Ruby::AST.callsites(String, :send)
transform = sites.transforms(:future, :existing)
transform = transform.thread_guard(:refine_dasherize)
```

```
# same performance impact as above
f << transform.wrap do |target, target_method_as_proc, *args, &block|
  if args[0] == :dasherize
    # invoke directly -> this is a normal callsite
    # -> gets wrapped by previous transform
    target.dasherize
  else
    target_method_as_proc.call(*args,&block)
  end
end
```

```
# above deals with .send()
# similar could be achieved with Module.prepend
# but callsite modification is invisible to the stack
# and to the module hierarchy
# it's a matter of picking the right tools for the right job
```

```
transforms = Ruby::AST.methods(D).transforms(:future)
```

```
# enable refinement for all methods in ClassB and downstack
transforms.enter do |method_name, *args, &block|
  # thread tokens should be a MultiSet to allow reentrancy
  Thread.add_token(:refine_dasherize)
end
transforms.exit do |method_name, return_value, *args, &block|
  Thread.remove_token(:refine_dasherize)
  return_value
end
```

```
class D
  def m1
    "a_b".dasherize
  end

  def m2
    "a_b".send(:dasherize)
  end
end
```

```
# transforms(:future) -> code gets patched
require "some_gem"
```

```
 future_transforms.each{|t| t.suspend}

 # code doesn't get patched anymore
 require "other_gem"
```

Is this complex? Yes
Is this verbose? Yes
Can you shoot yourself in the foot with it? Yes

This is intentional. It's a low-level API meant to provide as much
freedom to the programmer as possible. Higher-level abstractions (such
as refinements) can be built ontop of it. These abstractions could cover
many of the conflicting use-cases we have discussed in this thread.

Inheritance? A matter of module/method selectors
File scope? A matter of existing/future transform selectiveness
Super calls? You have full control over them, they're just callsites
OOP integration? Only when you want to go the extra mile

Actually, if we combine AST modification and Module.prepend we can
actually get down-stack refinements with complete OOP-integration if
needed. Assuming an .unprepend is also possible.

In fact, let me write this down:

case c) now with metaclass coercion:

```
module StringRefinement
  def downcase
    super + "x"
  end
end

Class E
  def m1
    "Foo".send(:downcase)
  end
end

sites = Ruby::Ast.methods(E).callsites(String)
transforms = sites.transform(:existing)

# an alternative to .wrap
# similar to method .enter/.exist. Except it's for callsites
transforms.before do |target,method_name, *args, &block|
  target.metaclass.send(:prepend, StringRefinement)
end
transforms.after do |target,method_name, return_value, *args, &block|
  target.metaclass.send(:unprepend, StringRefinement)
  return_value
end

E.new.m1 # => foox
```

I think such a fine-grained API is what we will need. No single
high-level API can cover all the use-cases provided for Refinements and
retain performance at the same time. Simply because it is too coarse.

#212 - 12/02/2012 12:29 AM - Anonymous

In message "Re: [ruby-core:50412] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Sat, 1 Dec 2012 04:54:20 +0900, "headius (Charles Nutter)" headius@headius.com writes:

|However, you go on to say that the example below should work, and I don't understand why:
|
|module R
| refine String do
| def foo; p :foo; end
| end
|
| module M
| refine(Array) { ... }
| "".foo
| end
| "".foo

|end
|"".foo
|
|Nothing has changed here; none of the foo calls are within refine blocks, but your example indicates the innermost foo should work. Why?

I don't think I did. I meant:

```
module R
refine String do
def foo; p :foo; end
end

module M
refine(Array) {
"".foo
}
end
end
using R
"".foo
```

|This shouldn't work anyway. Why would we look at refinements for Integer if Fixnum has overridden those methods? This seems completely anti-OO to me. Are you saying refinements can route completely around overridden methods without touching the class that overrides them?

If you think it's anti OO, you are too influenced by Java OO. Think
of CLOS's around methods. I want refinements to be method decorators.

```
                                                matz.
```

#213 - 12/02/2012 12:29 AM - Anonymous

In message "Re: [ruby-core:50419] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Sat, 1 Dec 2012 06:46:26 +0900, "enebo (Thomas Enebo)" tom.enebo@gmail.com writes:
|
|
|Issue #4085 has been updated by enebo (Thomas Enebo).
|
|
|Could someone make a summary of the current proposed refinements feature in the next couple of days once the churn of corner cases slows down a bit? I have been reading through this and I have three observations:

Shugo made summary still under development.

http://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec

#214 - 12/02/2012 01:10 AM - trans (Thomas Sawyer)

| If you think it's anti OO, you are too influenced by Java OO. Think
| of CLOS's around methods. I want refinements to be method decorators.

Ok. This is a very different sort of thing then, and really has nothing to do with safe monkey-patching. I remember you thinking the same sort of thing for :pre and :post hooks. Why is it you think this is such a good design? Is it just b/c Lisp did it? Honestly, how well has that really turned out for Lisp? I'm not so sure this is going to work out like you think it will. It has a much different effect on design (as I suspect Charles will soon explain).

#215 - 12/02/2012 01:34 AM - matz (Yukihiro Matsumoto)

trans (Thomas Sawyer) :pre :post etc. are covered by #prepend, and I am happy with it (although it's uglier than :pre).
The fundamental use cases of refinements (and monkey patching) are:

- adding new methods to existing class
- intercepting (and modifying) existing methods

For the former, since there's no existing method, no problem will occur.
For the latter, since refinements are independent from actual inheritance tree, application of refinements on methods can be very easily unpredictable.
Some (including you) claim that's the nature of "scoped monkey patching", but I think by making refined methods behave like CLOS's around methods, that kind of problems can be greatly reduced. Of course, you have to learn how refined methods work first. But it's the nature of all new features.

Matz.

#216 - 12/02/2012 11:20 AM - headius (Charles Nutter)

Anonymous wrote:

> |This shouldn't work anyway. Why would we look at refinements for Integer if Fixnum has overridden those methods? This seems completely anti-OO to me. Are you saying refinements can route completely around overridden methods without touching the class that overrides them?
>
> If you think it's anti OO, you are too influenced by Java OO. Think
> of CLOS's around methods. I want refinements to be method decorators.

There is, of course, the concept of around methods in AOP. I'm familiar with the concept. I don't believe the definition of refinements as being like CLOS around methods had been mentioned up to this point. Is that the essential definition you'd like us to use for refinements going forward?

Refinements are different from CLOS around methods in a number of crucial ways, though, as far as I understand them.

- CLOS does not look at the call site to determine which around methods to call. Refinements must be active only for specific call sites. This makes implementing :around methods considerably easier, since we're always looking for :around, :before, :after methods on the target hierarchy rather than in some external floating structure.
- As a necessary side effect of the above, :around methods apply to all calls equally. This avoids the readability challenges that refinements introduce. I'd love to see refinements go this way and not be derived from the calling scope.
- :around methods are attached to the class hierarchy, not to an unrelated structure. This makes searching for them *much* easier. It also makes them more predictable...every call hits the same :around logic in the same order.

The concept of :before, :after, and :around would be useful to introduce into Ruby. Refinements unfortunately conbine them with caller-scoped method lookup. Maybe what should go into 2.0 is just the :before, :after, :around capability without the caller-scoping?

If we're going to look at refinements as :around methods, then your requirements for "super" would have to change. In CLOS, all :around methods fire all the time in the same order. Let's go with the original example of decorating Integer#/. You are right that the around logic is picked up even against Fixnum in CLOS :around methods:

```
[1]> (defclass rinteger () ())
#
[2]> (defclass rfixnum (rinteger) ())
#
[3]> (defmethod div ((f rfixnum)) (print "div in fixnum"))
#)>
[4]> (defmethod div :around ((i rinteger))
(print "before div in integer")
(call-next-method)
(print "after div in integer")
)
#)>
[5]> (setq my-fix (make-instance 'rfixnum))
#
[6]> (div my-fix)

"before div in integer"
"div in fixnum"
"after div in integer"
```

I've already pointed out the biggest difference: this applies to all call sites everywhere, not just some of them. So we're diverging from refinements a lot already. However, if we continue...

Let's also define an around in rfixnum:

```
[7]> (defmethod div :around ((i rfixnum))
(print "before div in fixnum")
(call-next-method)
(print "after div in fixnum")
)
WARNING: The generic function # is being modified, but has already been called.
#)>
[8]> (div my-fix)

"before div in fixnum"
"before div in integer"
"div in fixnum"
"after div in integer"
"after div in fixnum"
```

The :around in rfixnum first, then the one in rinteger, then the actual div method. This is predictable behavior; all :around methods fire first, then :before methods, then the target method, then :after methods, and then control returns to the :around methods. This would require changes to the way methods are looked up in Ruby's hierarchy, but it would be easier than implementing refinements entirely outside the class hierarchy.

It's worth pointing out that the order in which these :around methods are defined has no bearing on when they're dispatched; they obey the OO hierarchy:

```
[14]> (defmethod div :around ((i rfixnum2))
(print "before div in fixnum")
(call-next-method)
(print "after div in fixnum")
)
#)>
[15]> (defmethod div :around ((i rinteger2))
(print "before div in integer")
(call-next-method)
(print "after div in integer")
)
#)>
[16]> (div (make-instance 'rfixnum2))

"before div in fixnum"
"before div in integer"
"div in fixnum"
"after div in integer"
"after div in fixnum"
```

Refinements define their ordering completely outside the object hierarchy, which is one of my concerns about them. In fact, if you want something that's scoped but mimics :around methods, my original idea to have the methods in the class hierarchy would make the most sense. It would also match the definition-order-indepenent sequence in which the refinements are fired.

The above could be modeled like this:

```
module RefineInteger
refine Integer do
def /(other)
puts "before / in RefineInteger"
result = super
puts "after / in RefineInteger"
result
end
end
end
```

At this point, the Integer class has been modified to recognize the above definition of "/" as an "around" method for the normal "/".

```
module RefineFixnum
refine Fixnum do
def /(other)
puts "before / in RefineFixnum"
result = super
puts "after / in RefineFixnum"
result
end
end
end
```

At this point, Fixnum has the above definition of "/" defined as an "around" method for the normal "/".

```
using RefineFixnum # the Fixnum refinement is now active, but not the Integer refinement
1/1 # prints => before in RefineFixnum, after in RefineFixnum

using RefineInteger # the Integer refinement is activated
1/1 # prints => before in RefineFixnum, before in RefineInteger, after in RefineInteger, after in RefineFixnum
```

However this does not combine well with many files all defining their own refinements. CLOS does not handle such a case either...if you define two "around" methods at the same level in the hierarchy, the new one takes over the old one.

```
[17]> (defmethod div :around ((i rfixnum2))
(print "new before div in fixnum")
(call-next-method)
(print "new after div in fixnum")
)
WARNING: The generic function # is being modified, but has already been called.
WARNING: Replacing method #)> in #
#)>
[18]> (div (make-instance 'rfixnum2))
```

```
"new before div in fixnum"
"before div in integer"
"div in fixnum"
"after div in integer"
"new after div in fixnum"
```

This is obviously very different from refinements, where we may have multiple files define their own refinements. If all are in scope, they fire in the order in which they're defined...which will be unpredictable depending on the sequence of those loads. If some or all of them are not in scope, they will not fire at all.

I am trying to understand what you envision refinements to be, but I'm having a very difficult time with it.

#217 - 12/02/2012 12:14 PM - headius (Charles Nutter)

Trying to think aloud how things should be structured. Hopefully this will be helpful to others. I base this on my understanding of refinements up to this point.

VIRTUAL HIERARCHY AS A REPRESENTATION OF REFINEMENTS

I may be starting to form a picture of how refinements are structured. I'll try to summarize a bit...let me know what I have wrong.

Given a simple class hierarchy:

```
class Foo
def hello; puts "hello in Foo"; end
end

class Bar < Foo
def hello; puts "hello in Bar"; super; end
end
```

We have a hierarchy like this: Bar < Foo < Object

Method lookup for normal unrefined Ruby proceeds against this hierarchy. First Bar's implementation of "hello" is found and called, then its super call finds and calls Foo's implementation of "hello".

```
Foo.new.hello # sees hierarchy Foo < Object
Bar.new.hello # sees hierarchy Bar < Foo < Object
```

Now we add a refinement:

```
module M1
refine Foo do
def hello
puts "before hello in M1"
super
puts "after hello in M1"
end
end
end
```

The refine call creates an anonymous module associated with the Foo class and containing a single method definition "hello" which acts around the primary "hello" implementation on Foo-related class hierarchies.

For purposes of lookup, a refined call site now sees a different hierarchy. Let's use the refinement and call a method:

```
using M1
Foo.new.hello # sees hierarchy [M1 refinement] < Foo < Object
Bar.new.hello # sees hierarchy [M1 refinement] < Bar < Foo < Object
```

Refinements are activated by the "using" method. Conceptually, when a refinement is brought into a scope, it is seen as being between the target object and its class, intercepting lookup at the call site.

So if normal call-site lookup works like this:

1. Get target object's class
2. Search class for method implementation

Refined call-site lookup works like this:

1. Get target object's class
2. Get the refinements that are active for this call site
3. Search the refinements for method implementation
4. Failing that, search the original class for method implementation

METHOD LOOKUP AGAINST REFINED HIERARCHIES

Where things start to get fuzzy for me is in step (3). I will summarize how I believe lookup is supposed to proceed

For each class in the target class's hierarchy, look for an active refinement for that class. Refinements on descendants will mask refinements on ancestors, but may super to them if both are active. Super from the innermost refinement returns to searching the normal class hierarchy.

In this case, the call-site-specific hierarchy above makes sense. With method tables in place we have:

For the call:

```
Bar.new.hello
```

The call site sees a structure like:

```
Foo
{:hello => Foo's hello}
Bar < Foo
{:hello => Bar's hello}
M1 refinement < Bar
{:hello => M1's hello}
```

In this case, it does look similar to prepend, but it's a virtual prepend that only lives at certain call sites. The behavior of "super" here starts to be a bit more clear, since M1's hello naturally supers into Bar's hello, even though M1 refines Foo.

Am I correct so far?

If we add a refinement to Bar:

```
module M2
refine Bar do
def hello; puts "before hello in M2"; super; puts "after hello in M2"; end
end
end
```

Now if we're using both refinements:

```
using M1
using M2 # order is important...it defines the virtual hierarchy
Bar.new.hello
```

The hierarchy seen at the "hello" call site looks like this: [M2 refinement] < [M1 refinement] < Bar < Foo

And the call proceeds as follows:
M2's hello supers to...
M1's hello supers to...
Bar's hello supers to...
Foo's hello

The virtual hierarchy looks this way for two reasons:

- M2 was used after M1, so it appears first in the search
- The classes that M2 and M1 refine both appear in the normal hierarchy, so both get searched

If we use the modules in a different order, we should see the M1 and M2 results reversed, since they are searched in most-recently-used order *ahead* of the hierarchy and independent of its structure:

```
using M2
using M1
Bar.new.hello
```

The virtual hierarchy here is: [M1] < [M2] < Bar < Foo

This would mean that the refinements on Foo fire before the refinements on Bar and M1's hello supers into M2's hello (which supers into Bar's hello). This means that the structure of the original hierarchy has no bearing on the ordering of refinements. The virtual hierarchy from refinements is always based on the order in which they are used in a given scope, regardless of what classes they refine.

Still correct?

REFINEMENTS ARE BOTTOM-HIERARCHY, SCOPE-LOCAL PREPENDS

If so, then it seems like refinements are essentially bottom-of-the-hierarchy prepends, searched for a given call site only if activated in the current scope and if the target object contains a refined class/module in its hierarchy. Conceptually, that's a bit more clear than other definitions so far.

This also explains the terminology "overlay modules" in Shugo's patch. The modules defined by a refinement are "overlaid" at the call site and searched before searching the original class hierarchy.

The process of looking up a normal direct method call at a refined call site, in more detail:

1. Get the target object's class
2. Get the list of refinements active for this call site
3. For each refinement (in order of "using"), look for the refined class or module in the target object's class hierarchy
4. If it appears in the hierarchy, search it for the method name in question
5. If the method exists, use it for the call (and ideally cache it in some way)
6. If the method in question does not appear on any of the refinements, continue searching the normal class hierarchy

Super lookup proceeds in much the same way, starting from the point in the virtual hierarchy where the method appears and looking up the virtual hierarchy from that point.

CALLS WITHIN REFINED METHODS ARE REFINED

What about other calls within refined methods?

```
class Chicken
def cluck
puts bok * 3
end
def bok; "bok"; end
end
```

Refining a method does not force down-stream calls to see it. Only call sites with that refinement active see the method.

```
module RefineChicken
refine Chicken do
def bok
"buck" + super
end
end
end
```

```
using RefineChicken
Chicken.new.cluck # => returns "bokbokbok" because Chicken's primary "cluck" method is not refined.
```

But if we refine both methods, the "bok" call site is also refined and sees the refined "bok" method:

```
module RefineChicken2
refine Chicken do
def cluck
puts bok * 2
end
def bok
"buck" + super
end
end
end
```

```
using RefineChicken2
Chicken.new.cluck # => "buckbokbuckbok"
```

The search logic for the "cluck" call sees the following hierarchy: [RefineChicken2] < Chicken < Object

The search logic for the "bok" call sees the same hierarchy, because it is made from within a scope where RefineChicken2 is active.

Hopefully this is all correct and describes the basic structure of refinements in a way people can understand. I'll give some thought to the troublesome cases from this thread and see how they should behave.

#218 - 12/02/2012 01:07 PM - headius (Charles Nutter)

PROBLEM: reflection from within a refined scope

Several folks have stated they believe reflection from within a refined scope should reflect the refinements. I do not agree, based on a definition of refinements as scope-local.

```
class Avocado
def tasty?
true
end
```

end

module RefineAvocado
refine Avocado do
def call_tasty
method(:tasty?).call
end

```
def tasty?
  false
end
```

end
end

The argument is that method() here should return the refined version of tasty? (the one that returns false). I don't see any justification for this.

The "method" method is defined on Object:

class Object
def method; ... end
end

At the point where "method" is called, exactly one refinement is active: the RefineAvocado module which adds "check_tasty" and overrides "tasty?". So "method" dispatches to Object#method.

Object#method is not refined, since it is not defined within a scope where refinements are active. It does not see refinements active in scopes earlier in the call stack, so it produces a reference to the original "tasty?" method.

What logic dictates that Object#method should now always be a refined method that has to inspect the stack?

If Object#method did reflect refinements active in the caller's scope, it would make it possible to get the refined method...but it would make it impossible to get the *original* method. And the same change has been proposed for all reflective access...the refinements basically make it impossible to inspect the original class, because they're in the way.

Giving Object#method and friends stack powers also makes them impossible to wrap. If you put your own code around Object#method, it will *never* see refinements because your wrapper's scope gets in the way. So if we made these class-hierarchy-related methods reflect refinements, you would no longer be able to safely wrap any of the following methods:

Symbol#to_proc
Object#send
Object#method
Object#methods
Object#respond_to?
Module#instance_method
Module#instance_methods
Module#public_instance_methods
Module#protected_instance_methods
Module#private_instance_methods
Module#method_defined?
Module#public_method_defined?
Module#private_method_defined?
Module#protected_method_defined?
Module#public_class_method
Module#private_class_method
Module#public_instance_method
Module#singleton_methods
Module#protected_methods
Module#private_methods
Module#public_methods
defined? logic for methods and super

And this may be just a start.

Instead, I propose that reflection of refinements be exposed directly.

Kernel#active_refinements => array of refinements active in the current scope, similar to Module#nesting

Object#refined_method(active_refinements, name)
=> returns a Method representing the refined method that would be found for the given list of refinements, or nil
Object#send_refined(active_refinements, name, *args)
=> sends the given name + args as though the specified refinements were active

I guess what I'm asking is that you consider a world where all of Ruby is implemented in Ruby, without stack-walking or magic, and don't make that world harder to achieve by adding magic to so many core methods. If people want a way to reflect from within a refined scope, give them those tools;

don't change the semantics of the existing tools and give them new powers to inspect the call stack.

PROBLEM: to_proc and other coercions in relation to refinements

Some folks have said they think to_proc should reflect refinements. I also disagree here.

```
class Lemur
def scratch
puts "itchy itchy"
end
end

module RefineLemur
refine Lemur do
def scratch
"scratchy scratchy"
end
end
end

using RefineLemur

ary = [Lemur.new, Lemur.new]
ary.each &:scratch
```

This last line is essentially the same as &(:scratch.to_proc)

to_proc can be defined like this, in Ruby:

```
class Symbol
def to_proc
return proc {|obj| obj.send self}
end
end
```

It should be apparent why to_proc should not reflect refinements: the send call it makes does not live within a refined scope. You are getting a new proc in hand with logic to send that symbol to any object it is passed. Refinements don't enter into it.

If we return to the definition of refinements as scope-local prepends, there's no way to justify making to_proc reflect refinements. The scope where refinements are active leads up to the to_proc and each calls, but NO FURTHER. The to_proc call operates independent of the refined scope and cannot see refinements. Making it "special", so it can see up the call stack, would mean it can't be wrapped anymore, can't be implemented in Ruby. You are giving superpowers to a single core method that NO RUBY CODE CAN MIMIC.

We should not make yet more core class methods that are impossible to wrap or write in Ruby.

PROBLEM: scoping at file level, module level, or some other level

The main benefit from making refinements file-scoped is simplifying the refined lookup process. If refinements must all be specified at the top level, then we have a clear set of active refinements at any given time. If they're activated within scopes, it can get confusing:

```
using M1 # M1 is active
module Blah
using M2 # M1, M2 are active?
module Bubble
using M3 # M1, M2, M3 are active?
"string".some_call ...
end
using M4 # M1, M2, M4 are active?
end
```

M1 is active

The overall effect on virtual-hierarchy method lookup is still basically the same. The refined modules are searched in most-recently-used order. The virtual hierarchy for the some_call call above would be:

[M3] < [M2] < [M1] < String < Object

If we implement it like Shugo's patch, the overlay modules are attached to the call sites as they are encountered, and only those overlays will ever be active at that call site.

So I guess I'm saying the file-level requirement simplifies some things, but doesn't really change anything conceptually.

PROBLEM: Matz's example showing String refinements active in a refine Array block

This example should not work:

```
module M1
refine String do
def foo; puts 'foo'; end
end

module M2
refine Array do
"string".foo
end
end
end
```

Within the M2 refinement, only one refinement is active: M2. M2 does live within M1, but M1's refinements are not active within the body of M1, and therefore they should not be active within the body of M2 or the refine Array block.

The virtual hierarchy at the point of the "foo" call looks like this because no active refinements affect String: String < Object

I would like to hear justification why the "foo" call above should succeed. It does not fit my mental model of refinements.

PROBLEM: module_eval

Several folks have claimed refinements will only be useful if they can be applied dynamically to module_eval blocks. However, this does not fit any reasonable definition of refinements thus far.

A block lives within a scope, and that scope may or may not have refinements active. You cannot change what refinements the block will see in the same way that you can't change what constants it will see. The following does not work:

```
module A
B = 1
end

A.module_eval { B } # => NameError: uninitialized constant B
```

And the following should not work either:

```
module A
refine String do
def foo; puts 'foo'; end
end
end

module B
using A
end

B.module_eval { "string".foo } # => NoMethodError
```

The foo call does not appear within a refined scope, and therefore it must never reflect refinements. I think it's just as important for this feature to know definitively when it *won't* happen as when it *will* happen, and it should NEVER be possible to force refinements on Other People's Code.

More to come as I think about stuff more. Is it possible for me to get access to edit the wiki page? I'd like to try to fill out more details (assuming I've got the details right).

#219 - 12/02/2012 02:05 PM - trans (Thomas Sawyer)

=begin
This is why refinements as decorators break Christmas present principle.

Rudolph is a library to light everything in red. Our developer, elf1, knows better then to monkey patch:

```
# lib/rudolph.rb
require 'ansi'
module Rudolph
class RedString < String
def to_s
ANSI.red(self)
end
end
def self.light!(string)
RedString.new(string.to_s)
end
end
```

Now we all know on Christmas Santa likes a lot of blinky bling, so elf2 decides to go all out.

```
# lib/christmas/refinements
require 'ansi'

module Christmas
module Refinements
refine String do
def to_s
ANSI.blink(self)
end
end
end
end
```

Awesome. Now he just needs to setup Santa to spread good cheer and use Rudolph to ensure a well lit way.

```
require 'christmas/refinements'

using Christmas::Refinements

module Christmas
class Santa
def spread_cheer!
puts "Merry Christmas!"
end
def on_rudolph!
puts Rudolph.light!("The Way!")
end
end
end
```

But uh oh! Why is rudolph blinky? He can't be blinky b/c he must clearly guide the way!

Okay so that is a really silly example. But imagine a more serious scenario using the same pattern, i.e. a 3rd party library has subclassed some base class. You of course have no idea that the developer even used said base class --his library is a black box, as it should be. For your app you decide to refine said base class. You also want to use the 3rd party library, but lo, it appears to be broken!
=end

#220 - 12/02/2012 02:42 PM - trans (Thomas Sawyer)

=begin

> "Refining a method does not force down-stream calls to see it. Only call sites with that refinement active see the method."

If that's the case then rudolph's message won't be blinky? But if so, then santa's cheer won't be blinky b/c #puts wasn't refined.

But if elf2 did:

```
module Christmas
class Santa
def spread_cheer!
puts "Merry Christmas!".to_s
end
def on_rudolph!
puts Rudolph.light!("The Way!").to_s
end
end
end
```

Then we are back to the original assertion.
=end

#221 - 12/02/2012 03:01 PM - headius (Charles Nutter)

trans (Thomas Sawyer) wrote:

> =begin
>
> > "Refining a method does not force down-stream calls to see it. Only call sites with that refinement active see the method."
>
> If that's the case then rudolph's message won't be blinky? But if so, then santa's cheer won't be blinky b/c #puts wasn't refined.

Correct. Your definition of Rudolph.light! does not occur in a refined context, so it is calling the normal String#to_s.

> But if elf2 did:
>
> module Christmas
> class Santa
> def spread_cheer!
> puts "Merry Christmas!".to_s
> end
> def on_rudolph!
> puts Rudolph.light!("The Way!").to_s
> end
> end
> end
>
> Then we are back to the original assertion.

If this is the entire file, none of these calls are refined. The first to_s will be String#to_s, and the second to_s will be RedString#to_s.

If there's a using Christmas::Refinements appears somewhere above this code, things are different. *Both* to_s calls will see the refined to_s (ANSI.blink version). Since that version does not super, both to_s's would blink, but neither would be red (since neither the original to_s nor RedString#to_s get called).

#222 - 12/02/2012 03:02 PM - headius (Charles Nutter)

headius (Charles Nutter) wrote:

> If there's a using Christmas::Refinements appears somewhere above this code, things are different. *Both* to_s calls will see the refined to_s (ANSI.blink version). Since that version does not super, both to_s's would blink, but neither would be red (since neither the original to_s nor RedString#to_s get called).

I should note that this is based on what I *think* matz and shugo envision for refinements today. This could certainly change in the future (or I could be wrong about what they envision).

#223 - 12/03/2012 05:53 AM - Anonymous

Hi guys,

Sorry to hijack the conversation, but I have been quickly reading the
thread and wow, it sounds like the specifications for this feature
changed at least a couple times. I was wondering if the core team
(Matz?) has been working on a readable spec that could be shared with
language implementors? Such a spec would be helpful (at least to me)
to understand the behaviors and how to implement them. I'm looking at
adding the 2.0 stuff in RubyMotion.

Sounds like RubySpec could be used for that?

Also, for what it counts, I have to agree with Charles that this
feature seems to be very ambitious for 2.0.

Regards.
Laurent

On Sun, Dec 2, 2012 at 7:02 AM, headius (Charles Nutter)
headius@headius.com wrote:

> Issue #4085 has been updated by headius (Charles Nutter).
>
> headius (Charles Nutter) wrote:
>
>> If there's a using Christmas::Refinements appears somewhere above this code, things are different. *Both* to_s calls will see the refined to_s (ANSI.blink version). Since that version does not super, both to_s's would blink, but neither would be red (since neither the original to_s nor RedString#to_s get called).
>
> ## I should note that this is based on what I *think* matz and shugo envision for refinements today. This could certainly change in the future (or I could be wrong about what they envision).
>
> Feature #4085: Refinements and nested methods

https://bugs.ruby-lang.org/issues/4085#change-34308

Author: shugo (Shugo Maeda)
Status: Assigned
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Category: core
Target version: 2.0.0

=begin
As I said at RubyConf 2010, I'd like to propose a new features called "Refinements."

Refinements are similar to Classboxes. However, Refinements doesn't support local rebinding as mentioned later. In this sense, Refinements might be more similar to selector namespaces, but I'm not sure because I have never seen any implementation of selector namespaces.

In Refinements, a Ruby module is used as a namespace (or classbox) for class extensions. Such class extensions are called refinements. For example, the following module refines Fixnum.

module MathN
refine Fixnum do
def /(other) quo(other) end
end
end

Module#refine(klass) takes one argument, which is a class to be extended. Module#refine also takes a block, where additional or overriding methods of klass can be defined. In this example, MathN refines Fixnum so that 1 / 2 returns a rational number (1/2) instead of an integer 0.

This refinement can be enabled by the method using.

class Foo
using MathN

```
def foo
  p 1 / 2
end
```

end

f = Foo.new
f.foo #=> (1/2)
p 1 / 2

In this example, the refinement in MathN is enabled in the definition of Foo. The effective scope of the refinement is the innermost class, module, or method where using is called; however the refinement is not enabled before the call of using. If there is no such class, module, or method, then the effective scope is the file where using is called. Note that refinements are pseudo-lexically scoped. For example, foo.baz prints not "FooExt#bar" but "Foo#bar" in the following code:

class Foo
def bar
puts "Foo#bar"
end

```
def baz
  bar
end
```

end

module FooExt
refine Foo do
def bar
puts "FooExt#bar"
end
end

end

module Quux
using FooExt

```
foo = Foo.new
foo.bar  # => FooExt#bar
foo.baz  # => Foo#bar
```

end

Refinements are also enabled in reopened definitions of classes using refinements and definitions of their subclasses, so they are *pseudo*-lexically scoped.

class Foo
using MathN
end

class Foo
MathN is enabled in a reopened definition.
p 1 / 2 #=> (1/2)
end

class Bar < Foo
MathN is enabled in a subclass definition.
p 1 / 2 #=> (1/2)
end

If a module or class is using refinements, they are enabled in module_eval, class_eval, and instance_eval if the receiver is the class or module, or an instance of the class.

module A
using MathN
end
class B
using MathN
end
MathN.module_eval do
p 1 / 2 #=> (1/2)
end
A.module_eval do
p 1 / 2 #=> (1/2)
end
B.class_eval do
p 1 / 2 #=> (1/2)
end
B.new.instance_eval do
p 1 / 2 #=> (1/2)
end

Besides refinements, I'd like to propose new behavior of nested methods.
Currently, the scope of a nested method is not closed in the outer method.

def foo
def bar
puts "bar"
end
bar
end
foo #=> bar
bar #=> bar

In Ruby, there are no functions, but only methods. So there are no right places where nested methods are defined. However, if refinements are introduced, a refinement enabled only in the outer method would be the right place. For example, the above code is almost equivalent to the following code:

def foo
klass = self.class
m = Module.new {
refine klass do
def bar

> puts "bar"
> end
> end
> }
> using m
> bar
> end
> foo #=> bar
> bar #=> NoMethodError
>
> The attached patch is based on SVN trunk r29837.
> =end
>
> --
> http://bugs.ruby-lang.org/

#224 - 12/03/2012 05:53 AM - zzak (Zachary Scott)

Hello Laurent,

On Sun, Dec 2, 2012 at 3:40 PM, Laurent Sansonetti
laurent.sansonetti@gmail.com wrote:

> changed at least a couple times. I was wondering if the core team
> (Matz?) has been working on a readable spec that could be shared with
> language implementors?

There is a wiki page:
http://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec

#225 - 12/03/2012 07:21 AM - trans (Thomas Sawyer)

headius (Charles Nutter) Thank you. I see how matz is conceiving of refinements now. Previous explanations as a means of safe monkey patching are not really accurate. It's not monkey patching. It is more akin to a macros --macros that intercept method calls. And thus I see how it avoids violation of modularity, b/c it doesn't effect any method that is not *explicitly* invoked within the scope of using the refinement.

So refinements, as it turns out, are rather a weak sauce. They CANNOT be used to "safely" inject behavior into pre-existing reusable classes. So I don't expect they will have much value for general system design. Primarily refinements will be useful for things like debugging and profiling.

Can they serve as a reliable replacement for monkey patching? To the extent that it is just to *add* a behavior, as opposed to changing a pre-existing behavior, then refinements might suffice. I think it remains to be seen how well they will really workout in this regard --whether a library like Facets or ActiveSupport should switch to a refinement only design, and whether they have any good use at all beyond that.

#226 - 12/03/2012 03:59 PM - headius (Charles Nutter)

I think I should point out that as of today, we have 20 days until code freeze for Ruby 2.0.0, and we are still trying to understand how refinements are going to work. I think this alone is evidence that the feature is still experimental, still contentious, still being designed. Unless we can come to some sort of reasonable consensus immediately, there will not be time to implement all changes and too little time to know if we've done the right thing. And if we haven't, code freeze will make it harder to recover.

I strongly urge matz to reconsider forcing this feature into 2.0 when it's not ready. I am not opposed to the feature in general, especially as we have narrowed its scope and as I have been able to better understand its implications. But currently, refinements are in a state of flux...and 2.0 needs to be settling down.

Let's move the feature to 2.1, possibly including an explicitly experimental form of refinements in the 2.0 release. I am willing to help continue this discussion for 2.1, so we can deliver the feature people actually want/need without being hasty.

#227 - 12/03/2012 05:01 PM - shugo (Shugo Maeda)

matz wrote:

> |This shouldn't work anyway. Why would we look at refinements for Integer if Fixnum has overridden those methods? This seems completely anti-OO to me. Are you saying refinements can route completely around overridden methods without touching the class that overrides them?
>
> If you think it's anti OO, you are too influenced by Java OO. Think
> of CLOS's around methods. I want refinements to be method decorators.

I missed it in your original post.
At first I thought you meant to make a subset of the current feature set of Refinements, but it's very different from the current behavior.
And once decided, it should not be changed in the future which of refinements and subclasses have priority.
I think it's hard to make an agreement within several days.

headius (Charles Nutter) wrote:

> Let's move the feature to 2.1, possibly including an explicitly experimental form of refinements in the 2.0 release. I am willing to help continue this discussion for 2.1, so we can deliver the feature people actually want/need without being hasty.

I agree with you, Charles. Refinements should not be included in 2.0, at least as an official feature.

#228 - 12/04/2012 06:33 PM - shugo (Shugo Maeda)

shugo (Shugo Maeda) wrote:

> > Let's move the feature to 2.1, possibly including an explicitly experimental form of refinements in the 2.0 release. I am willing to help continue this discussion for 2.1, so we can deliver the feature people actually want/need without being hasty.
>
> I agree with you, Charles. Refinements should not be included in 2.0, at least as an official feature.

Can I remove Refinements or make them experimental, Matz?

#229 - 12/06/2012 12:50 AM - matz (Yukihiro Matsumoto)

=begin
I have discussed with Shugo, and found out there was misunderstanding in my side.
So I withdraw the idea of refined methods as method decorators. So the refinement spec should be the subset of the current refinement implementation basically.

To repeat the difference from the current implementation

- "using" is only allowed at top level.
- refined methods are called only after "using".
- or within blocks given to "refine".
- if you pass the proc to "refine" e.g. refine(C,&b) refined methods may not be called from b. It's implementation dependent.
- refinements are not available in subclasses, nor in reopened classes/modules.
- refinements are not available from module_eval/class_eval.

Things left to the future decision.

- whether "include" make module to inherit refinement as well.
- way to combine "require" and "using", or to let refinements be available within a gem.

Matz.

p.s.
I strongly object to remove refinement from 2.0 but admit we may need to mark it experimental.
The final decision should be made within a few weeks.

=end

#230 - 12/06/2012 02:16 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I don't really like the idea of making something implementation dependent, specially when we can avoid it. Couldn't you just specify the desired behavior as the spec for the proc case?

#231 - 12/06/2012 08:00 AM - trans (Thomas Sawyer)

> I have discussed with Shugo, and found out there was misunderstanding in my side. So I withdraw the idea of refined methods as method decorators.

Wait, what? I finally understood your idea of refinements, and now it's not that any more?

Could you enlighten us on what Shugo explained? And these points you give are not very clear. It makes it sound as if they can't do much of anything. I.e. "not available in subclasses" --everything is a subclass of something. And "nor in reopened classes/modules" --isn't that every monkey patch?

The insanity and confusion of this. I swear, here there be ghettos!

#232 - 12/06/2012 11:48 AM - matz (Yukihiro Matsumoto)

=begin
trans (Thomas Sawyer), sorry for confusing.

The points are:

- I thought I explained how around method-like refinement behavior and he said it was OK, the truth is not.
- The implementation change for this behavior change is far bigger than I expected, probably won't be available before code freeze.
- And since this also requires performance impact (which I didn't expect neither), and we have no time to fix.
- "super" in the refined method will be hard to predict, thus hard to optimize.
- above facts made me give up around like methods.

Sorry for confusion.

"not available in subclasses" does not mean much, since we won't have class/module level "using" anymore.
That only means refinements are available based on lexical scope, not inheritance scope.

And "nor in reopened classes/modules" means you can not do the trick like:

```
# not in the refine scope
DB.find{
# symbol comparison added using module_eval
:age > 12
}
```

That's against the file-scope principle.

Matz.

=end

#233 - 12/06/2012 05:14 PM - trans (Thomas Sawyer)

matz (Yukihiro Matsumoto) Ok, thanks. That helps, but clarify for me: Are you saying that "above facts made me give up around like methods". Is that forever? Or only for this release? Will Ruby 2.1 make refinements around like methods? If yes, them maybe just wait til 2.1. If no, then could you explain behavior of refinements in class hierarchy?

#234 - 12/06/2012 05:41 PM - judofyr (Magnus Holm)

matz (Yukihiro Matsumoto) wrote:

> - if you pass the proc to "refine" e.g. refine(C,&b) refined methods may not be called from b. It's implementation dependent.

Hm. This is kinda a bummer, because it means we can't support both refinements and plain monkey-patching:

```
def patch(klass, &blk)
if respond_to? :refine
refine(klass, &blk)
else
klass.class_eval(&blk)
end
end
```

(BTW: How do I get code formatted properly here in Redmine?)

#235 - 12/06/2012 06:32 PM - matz (Yukihiro Matsumoto)

=begin
trans (Thomas Sawyer) No, this (non around) behavior will stay forever.

- Refined methods (methods defined in refinements) are only available after top-level "using".
- Not from called methods (i.e. no local rebinding).
- Refined method can be called on objects of subclass, unless the named method is redefined in the subclass.

```
class Foo
def foo; p :Foo; end
def bar; p :Foo; end
end

class Bar < Foo
def foo; p :Bar; super; end
end

module R
refine Foo do
def foo; p :R; super; end
def bar; p :R; end
end
```

end

using R
f = Foo.new
f.foo # refined / R -> Foo
f.bar # refined / R (no super)
b = Bar.new
b.foo # not refined Bar -> Foo
b.bar # refined R (no super)

Matz.

=end

#236 - 12/06/2012 08:10 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Wow, what a mess. It is really weird that Foo#foo (f.foo) prints "R" but Bar#foo (b.foo) doesn't once Bar inherits Foo and Bar#foo calls super.

It's very likely that I'll avoid using refinements as much as I can to avoid getting my head into trouble with all those weird rules.

It seems the only reason for the rule above is to avoid performance loss. It means that if somehow we figure out later how to avoid the performance hit we won't be able to do it because it would break compatibility :(

#237 - 12/06/2012 08:26 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Also, would you mind explaining why #using should be allowed only on top-level object?

I'd really prefer to limit the refinements to block scopes:

class MyController
def an_year_ago
using ActiveSupport::TimeCalculation
render json: { an_year_ago: 1.year.ago.to_i }
end

def unrelated_method
1.year # should raise NoMethodError in my opinion
end

def what_I_really_wanted
using ActiveSupport::TimeCalculation do
render json: { an_year_ago: 1.year.ago.to_i }
end
1.year # should raise NoMethodError
end
end

How do I format code on this Redmine fork by the way?

#238 - 12/06/2012 08:39 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Another example why I'd prefer local "using" instead of a global (file-scoped) one:

using Sequel::SmartSymbols do
DB[:some_table].where(:some_column > 3).first
end

I don't want symbol to have any special behavior here

Of course, for the case above I'd really prefer to be able to write something like:

Sequel.use_smart_symbols_in_queries = :enabled # in some config file
DB[:some_table].where{:some_column > 3}.first # in code around my application

But on the other hand I understand how complicated it would be to debug applications when the libraries can change object's behavior inside blocks passed to it. That is the reason why I wouldn't like Ruby to support the last example above even though it reads better. But I don't understand why the first example above would degrade performance for applications not using refinements.

#239 - 12/07/2012 12:02 AM - matz (Yukihiro Matsumoto)

rosenfeld (Rodrigo Rosenfeld Rosas),
If you don't understand it, and you want to avoid it, that's OK. Ruby is a good language without refinement.
But once you understand the rule, it seems the behavior is consistent.

The lesson we learned from AcriveSupport is that we don't care if methods added to the existing classes.
So effort to support "local refinement" seems not to be worth it.

Matz.

#240 - 12/07/2012 01:01 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Look, refinements could probably save Ruby from many drawbacks that currently exist. But I don't really understand what problem you're trying to solve with your definition of refinements.

Currently there are some gems that support the concept of extending Symbols so that you can use it like in the example above. The problem is that currently you can't use both Sequel and Squeel gems because both override the same symbol methods.

So, by using your definition of refinements I'd need to separate them by file so that I could use "using Sequel::SymbolExtensions" in a file and "using Squeel::SymbolExtensions" in another file. And I wouldn't ever be able to use both in a single method.

The problem is not that I don't understand it. I do. I just don't find its behavior consistent.

Ruby is a good language (not good enough, though) and my preferred one but it has lots of drawbacks and I'd really love to see its drawbacks being fixed (lack of proper thread support in MRI is the main one by the way). That is why I set up some time everyday to read all messages in the ruby-core mailing list and say my opinions on them when I find them relevant.

For this particular issue I'm just asking for the reasoning behind your design decisions so that I can understand your point of view before I can try to persuade you to go in another direction. But you haven't explained any reasons in your previous message. So I don't know how you expect me to understand the rule or to find it consistent if I don't understand the problem you're trying to solve. It would be helpful if you could provide some real-world example that you find refinements would help implementing.

Also, I didn't learn anything from ActiveSupport so I have no idea what you're talking about when you say "we don't care if methods are added to the existing classes". Particularly, I do care, specially when your project grows and you don't know what libraries added new methods to existing classes of if some built-in method has been overriden. It is pretty likely that conflicts will arise as the project grows and lots of libraries are used.

I value open classes and monkey patching but not as something to use as your daily programming tool. It should be used to fix some library until it is fixed main-stream in my opinion. Abusing from such language feature is not a good programming practice in any language.

I'm pretty sure others will agree with me but since Ruby doesn't provide library authors other options (like local refinements) they have a tradeoff and often they'll opt for monkey patching native classes (like symbols) even when they agree that those methods should exist only in certain contexts but Ruby doesn't provide them any feature to allow this to happen so they end up with the monkey patch approach.

This situation could be greatly improved if Ruby provided some feature like local refinements.

#241 - 12/07/2012 01:41 AM - matz (Yukihiro Matsumoto)

I haven't deny the future possibility to introduce your "local refinement". But we have to prepare working implementation in a week or so. We need more time to discuss about it.

What I meant by "consistent" is refinement apply order. What do you think the behavior inconsistent?

Matz.

#242 - 12/07/2012 01:57 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

What I find inconsistent specifically (and we wouldn't be able to fix it later without breaking backward compatibility) is the "super" behavior described in your example above (which I'm duplicating below):

```
class Foo
def foo; p :Foo; end
def bar; p :Foo; end
end

class Bar < Foo
def foo; p :Bar; super; end
end

module R
refine Foo do
def foo; p :R; super; end
def bar; p :R; end
end
end

using R

b = Bar.new
b.foo # not refined Bar -> Foo
b.bar # refined R (no super)
```

This is what I'd expect from the code above to find it consistent:

b.foo # p :Bar; p :R; p :Foo - this is what I'd expect something different than you, or maybe I misinterpreted you notation above?
b.bar # p :R - I think this is the same you expect reading your example above, just wanted to confirm

I can't understand how anything other than the above would be consistent from any perspective. Could you please explain the reason behind the behavior you want for "super" inside refinements definitions?

#243 - 12/07/2012 03:14 AM - trans (Thomas Sawyer)

matz (Yukihiro Matsumoto) Your example looks like you are trying to have it both ways. Sort of local but sort of not. I agree with rosenfeld (Rodrigo Rosenfeld Rosas) in that b.foo should behave with the refined Foo in place. using R has refined Foo and so everything effected by it should act as if Foo had been monkey patched --that's intelligible behavior. If this was the around method decorators that you wanted, then I'd understand. But since it is not and as you said it cannot be, there is no point to trying to "fake" it.

Sincerely, I implore your to consider: Refinements need a design-based spec, not an implementation-based spec. B/c all the later really means here is that you guys are winging it in order to squeeze it into the current feature freeze deadline. If you really want to include refinements in 2.0 then delay the freeze to get it right. I'm sure it could be done in short enough order, a week or even a five week delay won't be the end of the world. I am sure Rubyists every where will be much happier to know that you took the time to get it right, rather then rush it.

#244 - 12/07/2012 03:23 AM - Anonymous

In message "Re: [ruby-core:50641] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Fri, 7 Dec 2012 01:57:55 +0900, "rosenfeld (Rodrigo Rosenfeld Rosas)" rr.rosas@gmail.com writes:

|This is what I'd expect from the code above to find it consistent:
|
|b.foo # p :Bar; p :R; p :Foo - this is what I'd expect something different than you, or maybe I misinterpreted you notation above?
|b.bar # p :R - I think this is the same you expect reading your example above, just wanted to confirm

The point is we do not have local rebinding, that means refined method
is only available in lexical scope. In the example above, super in
Bar#foo is not in the refinement scope. That's the reason refined
method never called.

This is an artificial example, but in real use-case refinement should
be defined for subclasses as well, when a method is redefined in the
subclasses. That was original motivation for (abandoned) around
method-like refinement, but I decided it should be covered by
convention.

```
                                matz.
```

#245 - 12/07/2012 10:23 AM - The8472 (Aaron G)

On 06.12.2012 17:01, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> The problem is that currently you can't use both Sequel and Squeel gems because both override the same symbol methods.

That's not necessarily true. Squeel supports the extended-symbol as
legacy syntax for metawhere support, which you can turn off at load
time. It also provides a far better strategy:

Building an AST via BasicObject + instance_eval + method missing.

That way practically anything that's not a local variable can become a
DSL-keyword.

Date-patching to fixnums could be easily circumvented in DSLs too by
doing something like

```
dsl{in(15).days; after(15).days}
```

So really. DSLs should not be the issue here, if people would actually
design them properly.

Only library-level extensions to Strings/Arrays/Hash etc. such as those
of ActiveSupport should be considered a use-case for refinements, since
you don't use them in a very localized scope. You actually use them
throughout your whole library and don't want to be bothered to include
something into every file just to be able to use your utility method.

#246 - 12/07/2012 02:17 PM - shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

> I have discussed with Shugo, and found out there was misunderstanding in my side.
> So I withdraw the idea of refined methods as method decorators. So the refinement spec should be the subset of the current refinement implementation basically.

I've written down the new specification:

https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec

Please tell me if there's misunderstanding.
You hadn't describe the behavior when main.using is invoked in eval, so I guessed it.

The implementation hasn't been changed yet.

> p.s.
> I strongly object to remove refinement from 2.0 but admit we may need to mark it experimental.
> The final decision should be made within a few weeks.

I've made Refinements support a extension library named refinement.so.
When refinement.so is required, the following warning is shown:

/path/to/refinement.so: warning: Refinements are experimental, and the behavior may change in future versions of Ruby!

I hope the warning will be removed in the future, but it must be hard to remove it by the release of 2.0.

#247 - 12/07/2012 03:16 PM - matz (Yukihiro Matsumoto)

=begin
I have reviewed, and as far as I understand, it's correct, except that Module#eval should be Module#module_eval in one occation.

Matz.
=end

#248 - 12/07/2012 03:34 PM - shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

> I have reviewed, and as far as I understand, it's correct, except that Module#eval should be Module#module_eval in one occation.

I've fixed it. Thanks.

#249 - 12/07/2012 10:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-12-2012 23:17, The 8472 escreveu:

> On 06.12.2012 17:01, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:
>
>> The problem is that currently you can't use both Sequel and Squeel
>> gems because both override the same symbol methods.
>
> That's not necessarily true. Squeel supports the extended-symbol as
> legacy syntax for metawhere support, which you can turn off at load
> time. It also provides a far better strategy:
>
> Building an AST via BasicObject + instance_eval + method missing.
>
> That way practically anything that's not a local variable can become a
> DSL-keyword.
>
> Date-patching to fixnums could be easily circumvented in DSLs too by
> doing something like
>
> dsl{in(15).days; after(15).days}
>
> So really. DSLs should not be the issue here, if people would actually
> design them properly.
>
> Only library-level extensions to Strings/Arrays/Hash etc. such as
> those of ActiveSupport should be considered a use-case for
> refinements, since you don't use them in a very localized scope. You

> actually use them throughout your whole library and don't want to be
> bothered to include something into every file just to be able to use
> your utility method.

That was just an example justifying that local refinements can be quite
useful and that multiple libraries may want to override (or add) the
same methods to some core class but with different behaviors (thus
creating conflicts). I don't really have a need for Squeel (or
MetaWhere) and even for Sequel I disable all core extensions (Sequel
also provides this option) since I don't want any library to be adding
(or modifying) methods in core classes:

http://sequel.rubyforge.org/rdoc/files/doc/core_extensions_rdoc.html

but "where(:column.like('A%'))" reads better than "where(Sequel.like
:column, 'A%')" and I always alias Sequel as S so that I can write
"where(S.like :column, 'A%')".

I'm just saying that it would be great if we could support something
like "where{:column.like 'A%'}" without monkey patching symbols in all
contexts. But at the same time I don't want Ruby to support this because
it could make debugging much harder and it could degrade code
readability in many ways. So, as a trade-off, I'd like to be able to do
something like:

using Sequel::CoreExtensions do
records = DB[:some_table].
where(:some_column.like '%A').
except(:other_column < 3).
where(:another_one.in [3, 6]).
order(:sort_column.desc)
end

I know Sequel doesn't really support all extensions above, this is just
an example query why I think local refinements could be useful.

I much prefer the approach above instead of the DSL approach you
demonstrated.

#250 - 12/07/2012 10:43 PM - trans (Thomas Sawyer)

=begin
rosenfeld (Rodrigo Rosenfeld Rosas) I think the point was that a better designed API could do:

records = DB[:some_table].
where{some_column.like '%A'}.
except{other_column < 3}.
where{another_one.in [3, 6]}.
order{sort_column.desc}

=end

#251 - 12/08/2012 12:29 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Not all column names can be represented as method names. Or can they?

Even if they could I don't like this approach. Look, I currently maintain
an application that has some parts written in Grails, others in plain Java
and others in Rails. I can do things even more advanced than what you
suggested in Grails thanks to some features in Groovy.

But the problem begins when you have some local variable (or method) name
that happens to be the same as the column.

In such cases the DSL approach doesn't really help and may yield to
unexpected results (from a user POV)

#252 - 12/08/2012 01:02 AM - shugo (Shugo Maeda)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> I don't really like the idea of making something implementation dependent, specially when we can avoid it. Couldn't you just specify the desired behavior as the spec for the proc case?

In CRuby, it's possible to raise an ArgumentError if a given block is of a Proc.

PROC = Proc.new { .... }
module M
refine(String, &PROC) #=> ArgumentError
end

However, I'm not sure it's possible in other implementations such as JRuby.

#253 - 12/08/2012 03:48 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Charles will know for the JRuby case but if it currently doesn't differentiate blocks from procs (or lambdas) I guess they should be able to. If you think raising an ArgumentError for this case would be the best solution for now, then I'd much prefer to have this behavior than something that is implementation specific.

I don't really think Charles would prefer for this to be implementation specific and then get a JIRA issue complaining that it doesn't behave like in CRuby as several other issues there...

Developers don't read language specs (ECMAScript, etc - does Ruby have an official written spec?) so they consider that code that works is valid code. And they will complain if it doesn't work in another VM that claims to support the same language. So I'm afraid Ruby developers won't ever know that they are not supposed to use procs (or lambdas) as "refine" blocks unless the interpreter complains about it (like your ArgumentError proposal).

Charles, could you please help us on this subject?

#254 - 12/08/2012 04:36 AM - headius (Charles Nutter)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> So, by using your definition of refinements I'd need to separate them by file so that I could use "using Sequel::SymbolExtensions" in a file and "using Squeel::SymbolExtensions" in another file. And I wouldn't ever be able to use both in a single method.

Is this a use case we want to support? Being able to use multiple different monkey-patches in the same method body? I am skeptical.

> The problem is not that I don't understand it. I do. I just don't find its behavior consistent.
>
> Ruby is a good language (not good enough, though) and my preferred one but it has lots of drawbacks and I'd really love to see its drawbacks being fixed (lack of proper thread support in MRI is the main one by the way). That is why I set up some time everyday to read all messages in the ruby-core mailing list and say my opinions on them when I find them relevant.
>
> For this particular issue I'm just asking for the reasoning behind your design decisions so that I can understand your point of view before I can try to persuade you to go in another direction. But you haven't explained any reasons in your previous message. So I don't know how you expect me to understand the rule or to find it consistent if I don't understand the problem you're trying to solve. It would be helpful if you could provide some real-world example that you find refinements would help implementing.
>
> Also, I didn't learn anything from ActiveSupport so I have no idea what you're talking about when you say "we don't care if methods are added to the existing classes". Particularly, I do care, specially when your project grows and you don't know what libraries added new methods to existing classes of if some built-in method has been overriden. It is pretty likely that conflicts will arise as the project grows and lots of libraries are used.
>
> I value open classes and monkey patching but not as something to use as your daily programming tool. It should be used to fix some library until it is fixed main-stream in my opinion. Abusing from such language feature is not a good programming practice in any language.
>
> I'm pretty sure others will agree with me but since Ruby doesn't provide library authors other options (like local refinements) they have a tradeoff and often they'll opt for monkey patching native classes (like symbols) even when they agree that those methods should exist only in certain contexts but Ruby doesn't provide them any feature to allow this to happen so they end up with the monkey patch approach.
>
> This situation could be greatly improved if Ruby provided some feature like local refinements.

Patches accepted? :-)

Honestly, there are certain features that are so difficult to implement efficiently that no matter how useful they may be they're not worth the cost. I believe local, non-lexical refinements fall into this category due to the complexity of constantly checking whether a refinement is active

#255 - 12/08/2012 04:42 AM - headius (Charles Nutter)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> This is what I'd expect from the code above to find it consistent:
>
> b.foo # p :Bar; p :R; p :Foo - this is what I'd expect something different than you, or maybe I misinterpreted you notation above?

The super call in Bar#foo is not refined. Why would you expect it to call the refined version of Foo#foo? And what should other people see? Do you want the same unrefined super call site in an arbitrary file, possibly not yours, to see the refined call just because you refined it in your file? Isn't that basically the failure of monkey-patching to begin with?

> b.bar # p :R - I think this is the same you expect reading your example above, just wanted to confirm

This is correct. Bar does not override bar, so you see the refined version.

> I can't understand how anything other than the above would be consistent from any perspective. Could you please explain the reason behind the behavior you want for "super" inside refinements definitions?

I don't see why super should ever call the refined version. There's no refinements active in that scope.

#256 - 12/08/2012 04:43 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

> Honestly, there are certain features that are so difficult to implement efficiently that no matter how useful they may be they're not worth the cost. I believe local, non-lexical refinements fall into this category due to the complexity of constantly checking whether a refinement is active

When you say that it couldn't be implemented efficiently it is not clear to me if you mean that code not using refinements would have its performance affected by the existence of such a feature. If that is not the case, then I think it is ok for the developer to decide whether the performance hit should be acceptable or not.

#257 - 12/08/2012 04:53 AM - The8472 (Aaron G)

On 07.12.2012 16:23, Rodrigo Rosenfeld Rosas wrote:

> Not all column names can be represented as method names. Or can they?

Most columns should be possible, considering that even unicode method
names are valid.

For those cases where it's not possible there is **send** in
BasicObject. Or you can manually generate the AST-Object that would
normally be spawned by method_missing. Or you could have a custom helper
method. Or you could just call method_missing directly.

And before you say that's ugly, compare the following:

a)

```
Foo.query{__send__("`illegal_method_name") == bar}
```

b)
using SomeDSLRefinement do
Foo.query(:"`illegal_symbol_name" => :bar)
end

Personally I consider the first one more elegant, concise and
expressive. Especially since you can actually write ruby code in the
block and thus do things dynamically.

> Even if they could I don't like this approach. Look, I currently
> maintain an application that has some parts written in Grails, others in
> plain Java and others in Rails. I can do things even more advanced than
> what you suggested in Grails thanks to some features in Groovy.
>
> But the problem begins when you have some local variable (or method)
> name that happens to be the same as the column.

Local methods are not a problem with an instance_eval'd block on
BasicObject. Local variables may conflict, but you have control over
them since they are by definition *local*.

> In such cases the DSL approach doesn't really help and may yield to
> unexpected results (from a user POV)

Those "problems" are far more benign than the havok sometimes caused by

monkey patching and usually result from a lack of understanding of the
employed metaprogramming methods. I.e. it's a problem that can be fixed
simply by increasing the user's knowledge through good documentation.

So really, i find this approach to making DSLs much cleaner than
patching around in core objects.

Of course monkey patching does have its place, I'm not going to deny
that. But DSLs shouldn't be the primary use-case for it. Glueing
together two libraries that don't interact nicely with each other or or
providing widely used utility methods throughout a whole gem/application
namespace would be far more important in my opinion.

#258 - 12/08/2012 04:55 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Charles I have read your arguments about super and refinements behavior above and now I understand that the expected behavior from Matz and you also makes sense. It is just that is very hard to me to understand what would make most sense without any real application code usage and I can't really think of one from the top of my head that would involve "super" calls.

#259 - 12/08/2012 05:02 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Aaron, I see your point but often my queries are much more complex than a single mention to some column. So if I have to repeat an unrepresentable column multiple times I'd prefer the block approach.

When I mentioned the DSL issues I did it from my previous experience with Grails. In Grails you can bind the params arguments as the controller's method's arguments. So, consider this:

class SomeController {
def someAction(String name) {
MyDomainClass.find { name == name} // WTF?! I want to compare the "name" column to the value of the "name" param
}
}

I know this isn't possible in Ruby (the params binding feature). But what if you want to compare "name" column to the result of a call to the "name" local (or inherited) method?

#260 - 12/08/2012 05:15 AM - jeremyevans0 (Jeremy Evans)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Aaron, I see your point but often my queries are much more complex than a single mention to some column. So if I have to repeat an unrepresentable column multiple times I'd prefer the block approach.
>
> When I mentioned the DSL issues I did it from my previous experience with Grails. In Grails you can bind the params arguments as the controller's method's arguments. So, consider this:
>
> class SomeController {
> def someAction(String name) {
> MyDomainClass.find { name == name} // WTF?! I want to compare the "name" column to the value of the "name" param
> }
> }
>
> I know this isn't possible in Ruby (the params binding feature). But what if you want to compare "name" column to the result of a call to the "name" local (or inherited) method?

Use parentheses to tell ruby to do a method call even if there is a local variable with the same name:

def some_action(name)
MyDomainClass.find{name() == name}
end

Jeremy

#261 - 12/08/2012 05:29 AM - The8472 (Aaron G)

On 07.12.2012 21:02, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Issue #4085 has been updated by rosenfeld (Rodrigo Rosenfeld Rosas).
>
> Aaron, I see your point but often my queries are much more complex than a single mention to some column. So if I have to repeat an unrepresentable column multiple times I'd prefer the block approach.
>
> When I mentioned the DSL issues I did it from my previous experience with Grails. In Grails you can bind the params arguments as the controller's method's arguments. So, consider this:

> class SomeController {
> def someAction(String name) {
> MyDomainClass.find { name == name} // WTF?! I want to compare the "name" column to the value of the "name" param
> }
> }
>
> I know this isn't possible in Ruby (the params binding feature). But what if you want to compare "name" column to the result of a call to the "name" local (or inherited) method?

Named parameters? Depends. If you can splat them I would simply suggest moving the thing to a different method which won't suffer from scoping issues. But really, that's a Groovy-issue in so far that it gives local variable names (method argument names essentially are local vars) a non-local significance which robs you of the freedom of renaming them as you desire without breaking code.

But there are solutions. Remember that it's instance_eval'd:

```
  MyDomainClass.find { self.name == name}
```

Alternatively you can teach the DSL some smartness and switch between call and instance_eval based on arity:

```
  MyDomainClass.find {|d| d.name == name}
```

There are many ways to solve this problem without polluting any external object.

> So if I have to repeat an unrepresentable column multiple times I'd prefer the block approach.

Block approach is a bad name for this, since both approaches are using blocks ;) Anyway, this problem can be solved too:

```
Foo.dsl do
  col = __send__("illegal_name")
  col.eq(coalesce(id,title)) | col.like("%bar%")
end
```

In fact, squeel provides an even more elegant solution. Since column names often are escaped with the ` character and that's a valid method name in ruby it provides literals that way:

```
MyModel.where{other_table.`literal_name` == "bar"}
```

You're really bringing up edge cases here for which there are multiple solutions. There is no necessity to monkey-patch Symbol only to build some DSLs, really.

#262 - 12/08/2012 06:23 AM - headius (Charles Nutter)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> > Honestly, there are certain features that are so difficult to implement efficiently that no matter how useful they may be they're not worth the cost. I believe local, non-lexical refinements fall into this category due to the complexity of constantly checking whether a refinement is active
>
> When you say that it couldn't be implemented efficiently it is not clear to me if you mean that code not using refinements would have its performance affected by the existence of such a feature. If that is not the case, then I think it is ok for the developer to decide whether the performance hit should be acceptable or not.

That is definitely the case for some definitions of refinements (such as those that support refining code in an already-created proc).

#263 - 12/08/2012 12:15 PM - shugo (Shugo Maeda)

- Assignee changed from matz (Yukihiro Matsumoto) to shugo (Shugo Maeda)

shugo (Shugo Maeda) wrote:

> I've written down the new specification:
>
> https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec

> Please tell me if there's misunderstanding.
> You hadn't describe the behavior when main.using is invoked in eval, so I guessed it.
>
> The implementation hasn't been changed yet.

I've implemented the new specification in the SVN trunk. Please try it.

There are some considerations remained:

- Should not send, method, respond_to? use refinements? I've changed Symbol#to_proc not to use refinements, but it might be better these three methods to use refinements.
- Should Module#include inherit refinements?
- Should Module#refinements be removed?
- In CRuby, refine(klass, &proc) raises an ArgumentError. Should it be the spec?

#264 - 12/08/2012 01:53 PM - Anonymous

In message "Re: [ruby-core:50681] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Sat, 8 Dec 2012 12:15:31 +0900, "shugo (Shugo Maeda)" redmine@ruby-lang.org writes:

|There are some considerations remained:
|
|* Should not send, method, respond_to? use refinements?
| I've changed Symbol#to_proc not to use refinements, but it might be better these three methods to use refinements.

For 2.0, any indirect method access need not to honor refinements.

|* Should Module#include inherit refinements?

Not for 2.0.

|* Should Module#refinements be removed?

Yes, it's not included in the latest spec.

|* In CRuby, refine(klass, &proc) raises an ArgumentError. Should it be the spec?

I guess so, but it's not worth to overhaul for JRuby and Rubinius.

```
                              matz.
```

#265 - 12/08/2012 10:24 PM - shugo (Shugo Maeda)

matz wrote:

> |* Should not send, method, respond_to? use refinements?
> | I've changed Symbol#to_proc not to use refinements, but it might be better these three methods to use refinements.
>
> For 2.0, any indirect method access need not to honor refinements.

Does "need not" mean an implementation may honor refinements?
Or does it mean just any indirect method access shall not honor refinements?

> |* Should Module#include inherit refinements?
>
> Not for 2.0.

I see.

> |* Should Module#refinements be removed?
>
> Yes, it's not included in the latest spec.

The spec at https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec has a placeholder for Module#refinements, but I'll remove it.

> |* In CRuby, refine(klass, &proc) raises an ArgumentError. Should it be the spec?
>
> I guess so, but it's not worth to overhaul for JRuby and Rubinius.

If it's difficult in other implementations to raise an ArgumentError, the behavior should be unspecified instead of implementation-defined because "unspecified" implies that an error might occur.

#266 - 12/09/2012 12:23 AM - Anonymous

In message "Re: [ruby-core:50694] [ruby-trunk - Feature #4085] Refinements and nested methods"
on Sat, 8 Dec 2012 22:24:23 +0900, "shugo (Shugo Maeda)" redmine@ruby-lang.org writes:

|matz wrote:
|> For 2.0, any indirect method access need not to honor refinements.
|
|Does "need not" mean an implementation may honor refinements?
|Or does it mean just any indirect method access shall not honor refinements?

It means in 2.0, those methods DO NOT treat refinements.

|> I guess so, but it's not worth to overhaul for JRuby and Rubinius.
|
|If it's difficult in other implementations to raise an ArgumentError, the behavior should be unspecified instead of implementation-defined because "unspecified" implies that an error might occur.

Yes, that what I mean.

```
                                matz.
```

#267 - 12/10/2012 08:14 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

headius (Charles Nutter) wrote:

> rosenfeld (Rodrigo Rosenfeld Rosas) wrote:
>
> > > Honestly, there are certain features that are so difficult to implement efficiently that no matter how useful they may be they're not worth the cost. I believe local, non-lexical refinements fall into this category due to the complexity of constantly checking whether a refinement is active
> >
> > When you say that it couldn't be implemented efficiently it is not clear to me if you mean that code not using refinements would have its performance affected by the existence of such a feature. If that is not the case, then I think it is ok for the developer to decide whether the performance hit should be acceptable or not.
>
> That is definitely the case for some definitions of refinements (such as those that support refining code in an already-created proc).

If I understand correctly this isn't the case for my local refinements request, right?

But it doesn't matter anyway as I have given up on the idea of local refinements as well (more on that soon in a separate comment).

#268 - 12/10/2012 08:48 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

jeremyevans0 (Jeremy Evans) wrote:

> Use parentheses to tell ruby to do a method call even if there is a local variable with the same name:
>
> def some_action(name)
> MyDomainClass.find{name() == name}
> end

What if you want to compare to a local (or inherited) method named "name" call result? I don't buy this argument, but what convinced me was actually Aaron's response that enlighted me.

Before his suggestion, I was considering an approach like used in many JavaScript libraries, like underscore, prototype, jQuery and RubyJS, i.e., by wrapping the object into another decorated one:

_(:some_column) > 4

But then Aaron suggested this simple approach:

MyDomainClass.find {|d| d.name == name}

I almost liked it, but I don't like the idea of dealing with several edge cases... For instance, what if the column is named "class"? But then, an approach like this would be great for me:

MyModel.find{|d| d[:class].ilike('c%') | (d[:created_at] < d[2].years.ago) }

Hence there is no need for local refinements for use cases like this in my opinion.

Alternatively, if you don't want to include helpers such as ActiveSupport time conversion modules in that dsl object, you could use a global one (assuming Numeric isn't patched) in a way similar to the approach used by the JS libraries mentioned above: _(2).years.ago.

Thank you for your input :)
Cheers

#269 - 12/11/2012 04:22 PM - headius (Charles Nutter)

Anonymous wrote:

> In message "Re: [ruby-core:50694] [ruby-trunk - Feature #4085] Refinements and nested methods"
> on Sat, 8 Dec 2012 22:24:23 +0900, "shugo (Shugo Maeda)" redmine@ruby-lang.org writes:
>
> |matz wrote:
> |> For 2.0, any indirect method access need not to honor refinements.
> |
> |Does "need not" mean an implementation may honor refinements?
> |Or does it mean just any indirect method access shall not honor refinements?
>
> It means in 2.0, those methods DO NOT treat refinements.

(referring to #send, #method, #respond_to?)

I think it's good to leave these free of refinements for the 2.0 experiment. We'll see if anyone misses the modified versions.

> |> I guess so, but it's not worth to overhaul for JRuby and Rubinius.
> |
> |If it's difficult in other implementations to raise an ArgumentError, the behavior should be unspecified instead of implementation-defined because "unspecified" implies that an error might occur.
>
> Yes, that what I mean.

I don't think it will be, but I have to see. I'm pretty sure we know once a block has turned into a proc.

I'll try to get the new specification implemented in JRuby this week as well, so it can be available for experimenting there too.

#270 - 12/15/2012 11:55 PM - sonysantos (Sony Santos)

Sorry, I'm arriving late (and I didn't read all discussion yet), but I've heard the ruby core team is looking for feedback on refinements, and I feel I must give my 2 cents.

What if refinements were truly, strictly lexically scoped? Nothing more than that? No affected subclasses, no module_eval issues, etc.

I think:

- it's enough for isolating conflicting libraries with core extensions (the target for refinements), and
- it breaks down all complexity of refinements, both for implementing as for programming, reading code etc.

I'm not a great or experienced Ruby programmer, but I made an emulation of how it would look:

http://rubychallenger.blogspot.com.br/2012/12/refinements-in-ruby-ingenuous.html

It's basic usage is:

module Camelize
refine String do
def camelize
dup.gsub(/_([a-z])/) { $1.upcase }
end
end
end

class Foo
Camelize.enable   # it could be "using Camelize", but I found it harder to implement by now

def camelize_string(str)
str.camelize
end

Camelize.disable
end

f = Foo.new
puts f.camelize_string('abc_def_ghi')   #=> abcDefGhi

it works because str.camelize is called from an enabled range for Camelize in this file.

class Bar < Foo
def camelize_and_join1(str_ary)
str_ary.map {|str| camelize_string(str) }.join(',') #=> ok, same reason above
end

def camelize_and_join2(str_ary)
str_ary.map {|str| str.camelize}.join(',')   #=> NoMethodError, unless I was "using Camelize" in Bar
end

# I don't see why I would want the Foo's "using Camelize" be valid here. I don't need that!
# If I need, I just add "using Camelize" again! Explicitly! Lexically! No more surprises in subclasses! KIS
end

sa = %w(abc_abc def_def ghi_ghi)

b = Bar.new
puts b.camelize_and_join1(sa)   #=> abcAbc,defDef,ghiGhi
puts b.camelize_and_join2(sa)   #=> NoMethodError

Since the call to String#camelize isn't in a enabled range for Camelize in this file.

So here is my suggestion: truly lexically scoped "using module". Simple as that. I guess we don't need more than that.

#271 - 01/07/2013 12:28 PM - shugo (Shugo Maeda)

- Status changed from Assigned to Closed

I've implemented all features requested by Matz, so I close this ticket.

The current spec is described at https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec.

If you have any request for Refinements, please file a new ticket.
However, the feature set of Ruby 2.0.0 has already been frozen, so the spec won't be changed in Ruby 2.0.0 unless otherwise permitted by Matz or Endoh-san.

Files

refinement-r29837-20101124.diff	70.3 KB	11/24/2010	shugo (Shugo Maeda)
control_frame_change-r29944-20101127.diff	31.5 KB	11/27/2010	shugo (Shugo Maeda)
refinements-r29944-20101127.diff	33.8 KB	11/27/2010	shugo (Shugo Maeda)
nested_methods-r29944-20101127.diff	8.65 KB	11/27/2010	shugo (Shugo Maeda)