Ruby - Feature #6594

Integrated Functor

06/15/2012 09:02 AM - trans (Thomas Sawyer)

Status:	Assigned
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	

Description

I know the developers meeting is coming up so I'd like to get a few ideas I've had sitting in the wings out in the air before then.

One the more useful is the idea of integrating Functors directly into the language. "Functor" is the term I use for "higher-order function".

I blogged about this idea and you can read it here: http://trans.github.com/2011-09-07-ruby-heart-higher-order-functions/

The super short version is this:

```
def f => op, arg
  arg.send(__op__, arg)
end

f + 3  #=> 6
f * 3  #=> 9
```

Another example:

```
class String
  def file => op, *args
    File.send(__op__, self, *args)
  end
end

"README.rdoc".file.mtime  #=> 2012-06-14 12:34:45 -0400
```

I'm using => as means of indicating a higher-order function. Of course another syntax could be used if this won't fly. The important thing is the idea of higher-order functions being integrated directly into the language. Doing this without that integration requires the creation of an intermediate object for each call which is very inefficient.

History

#1 - 06/15/2012 01:31 PM - shyouhei (Shyouhei Urabe)

-1.

trans (Thomas Sawyer) wrote:

> Doing this without that integration requires the creation of an intermediate object for each call which is very inefficient.

This should be fixed first. To introduce a new syntax to cover up a ruby's bug is definitely a bad idea.

#2 - 06/15/2012 03:19 PM - trans (Thomas Sawyer)

@shyouhei (Shyouhei Urabe) Fixed? How is there a "ruby's bug"?

#3 - 06/15/2012 03:45 PM - matz (Yukihiro Matsumoto)

If I understand correctly, if I define

```
def f => op, arg
end
```

then

```
f + a
```

should be parsed as

```
f.=>(:+, a)
```

when f is a functor, and as

```
f.+(a)
```

otherwise. Correct?

If so, changing semantics according to dynamic type is against my design policy.

#4 - 06/15/2012 04:18 PM - shyouhei (Shyouhei Urabe)

trans (Thomas Sawyer) wrote:

> @shyouhei (Shyouhei Urabe) Fixed? How is there a "ruby's bug"?

@trans You wrote "creation of an intermediate object ... is very inefficient". That thing should be lightweight, or should completely be skipped, without introducing such new syntax.

#5 - 06/15/2012 05:14 PM - trans (Thomas Sawyer)

@matz (Yukihiro Matsumoto) Nay. I guess my attempt at a brief synopsis was not a good idea.

Understanding Functor class would probably help best first. See https://github.com/rubyworks/facets/blob/master/lib/core/facets/functor.rb

Also, I realized I am not using 100% right terminology. Really I should be saying "Higher Order Messaging".

Here are some resources about this idea:

- http://en.wikipedia.org/wiki/Higher_order_message
- http://kbullock.ringworld.org/2007/03/26/higher-order-messaging/

Here is the link to my blog post: http://trans.github.com/2011-09-07-ruby-heart-higher-order-functions/

#6 - 06/15/2012 10:13 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

@trans it really helps attaching an useful example from a real world situation where functors would make it easier to implement some feature than using current Ruby features.

Good luck on trying to convince people to spend their time learning a new concept without first getting them interested on this new feature by providing a really great example usage.

#7 - 06/16/2012 07:25 AM - trans (Thomas Sawyer)

@resenfeld Well, I thought the examples I provided were pretty good ones. The links I provided also give some examples, though one has to mentally convert their implementations. If you'd like some others:

```
$ grep -R "Functor" facets/lib
lib/core/facets/symbol/as_s.rb:    Functor.new do |op, *a|
lib/core/facets/enumerator/fx.rb:    Functor.new(&method(:fx_send).to_proc)
lib/core/facets/hash/recursively.rb:    Functor.new do |op, &yld|
lib/core/facets/hash/data.rb:    Functor.new do |op, *a|
lib/core/facets/kernel/not.rb:    Functor.new(&method(:not_send).to_proc)
lib/core/facets/kernel/eigen.rb:    Functor.new do |op,*a,&b|
lib/core/facets/kernel/ergo.rb:    @_ergo ||= Functor.new{ nil }
lib/core/facets/kernel/try.rb:      Functor.new{ nil }
lib/core/facets/kernel/respond.rb:      Functor.new(&method(:respond).to_proc)
lib/core/facets/enumerable/ewise.rb:    Functor.new do |op,*args|
lib/core/facets/enumerable/per.rb:      Functor.new do |enumr_method, *enumr_args|
lib/core/facets/enumerable/per.rb:        Functor.new do |op, *args, &blk|
lib/core/facets/enumerable/per.rb:        Functor.new do |enumr_method, *enumr_args|
lib/core/facets/enumerable/per.rb:          Functor.new do |op, *args, &blk|
lib/core/facets/enumerable/accumulate.rb:    Functor.new do |op, *args|
lib/core/facets/enumerable/accumulate.rb:    Functor.new do |op, *args|
lib/core/facets/string/file.rb:    Functor.new(&method(:file_send).to_proc)
lib/core/facets/module/method_space.rb:      Functor.new do |op, *args|
```

And there would be more, but in some cases I opted against it b/c of the efficiency issue mentioned or b/c I created a specialized Functor class instead --mainly to keep support for 1.8.6- b/c Procs couldn't take blocks as arguments before.

#8 - 06/17/2012 12:06 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

@trans I wasn't claiming that it wasn't possible for those interested on the feature to find real-case examples about it. I was just warning you that some people won't want to spend their time trying to find such examples.

In the other hand if you take some time to choose a single example that you find to be a great one that you could write a Ruby code using the current syntax to show how to implement something and then compare to your proposed new syntax it might help getting the interest of others that are not currently interested enough on this feature to just start learning about it from other links. (Wow, I guess I've just written the biggest sentence I've ever written - sorry for that :P)

#9 - 06/18/2012 10:08 AM - prijutme4ty (Ilya Vorontsov)

I agree that high-order messaging can be very useful. But functor concept and usage is rather complex that makes it hard to understand in which real cases it's useful (where one can't stick to another way to do the same thing). Can you provide really cool usage of functor class?
If you want to easy build constructs like:
[0,1,2,3,4].having.succ < 3 (from article you mentioned),
I suggest you simplier way of doing so (my realization of such behavior
https://github.com/prijutme4ty/bioinform/blob/master/lib/bioinform/support/callable_symbol.rb):

```
[0,1,2,3,4].select(&:succ.() < 3)
```

or even so:

```
256.times.select(&:to_s.(2).size == 4)    # ==> [8,9,10,11,12,13,14,15]
```

and I can chain any calls in proc without any additional code for selectors etc. It's also HOM, of course. I hope one day Matz would eliminate dot before brackets of call method - then syntax'd be even more concise.

#10 - 06/19/2012 09:08 PM - trans (Thomas Sawyer)

@prijutme4ty Your link appears to be broken, but I am glad to see your interest on HOM. I would need to know more about your callable symbols idea to know if it is HOM like Functor. My impression from your example is that it might be a type of call recorder (akin to a Spy).

I am surprised that you think Functor seems complex, when basic model is simply:

```
class Functor
  def initialize(&function)
    @function = function
  end
  def method_missing(op, *args, &blk)
    @function.call(op, *args, &block)
  end
end
```

For one example, probably the first I ever made, is #every. I wrote about it at
https://groups.google.com/forum/?fromgroups#!topic/ruby-talk-google/zW_IDKq754A

#11 - 06/30/2012 06:22 AM - prijutme4ty (Ilya Vorontsov)

trans (Thomas Sawyer) wrote:

> @prijutme4ty Your link appears to be broken, but I am glad to see your interest on HOM. I would need to know more about your callable symbols idea to know if it is HOM like Functor. My impression from your example is that it might be a type of call recorder (akin to a Spy).
>
> I am surprised that you think Functor seems complex, when basic model is simply:
>
> ```
> class Functor
>   def initialize(&function)
>     @function = function
>   end
>   def method_missing(op, *args, &blk)
>     @function.call(op, *args, &block)
>   end
> end
> ```
>
> For one example, probably the first I ever made, is #every. I wrote about it at
> https://groups.google.com/forum/?fromgroups#!topic/ruby-talk-google/zW_IDKq754A

Sorry for a long delay with my reply. Here's a correct link to my code
https://github.com/prijutme4ty/bioinform/blob/master/lib/bioinform/support/callable_symbol.rb

Yes, my code is chain recorder in some sense. But really all examples of use functor are just the same simple one-line method replacement. My approach to this #every method looks like

```
[1,2,3].map &:+.(3)   #=> [4,5,6]
[1,2,3].map &:*.(3)   #=> [3,6,9]
```

and you can see, I don't create any additional like #every, I just use already existing methods.

Example with upcase can be easily written in core ruby

```
words = ["hello", "world"]
words.each &:upcase! # or words.map &:upcase
words  #=> ["HELLO", "WORLD"]
```

Can you give an example where creating Functor object really makes things easier. I understand that concept can be really very powerful, but not in such simple examples.
I've called Functor complex only because calling #send method always looks like a trick. I think that only libraries should use methods like #send and other metaprogramming techniques, but programmer as long as possible should stay away from using metaprogramming. What to me, methods like #every can be defined in a gem (but not in "user-code", so this shouldn't be such a usual task) so introducing new syntax is overkill unless you provide really fine cases where functor can be used in "user-code" (not for only a dozen of methods like #every).

#12 - 07/01/2012 05:27 AM - trans (Thomas Sawyer)

@prijutme4ty I think you last paragraph actually makes it quite clear why I make this proposal.

You say that "only libraries should use methods like #send and other metaprogramming techniques". But you see meta-programming is the *only* way to implement this presently. What I am proposing is to address precisely that --a means of getting rid of having to use #send and "meta-programming". Then you say, "methods like #every can be defined in a gem". But that hits the very problem of defining any HOM, such as #every: it can't be done *efficiently* in pure Ruby. There is simply no way to do it b/c one has to create this (usually) extraneous Functor object to act as the intermediary. So #every is actually a very good example precisely b/c it is so simple.

I think your callable symbol idea is a good one, but it is more limited than Functor, which is full HOM. For instance how would you handle Enumerable#accumulate (http://rdoc.info/github/rubyworks/facets/master/Enumerable#accumulate-instance_method)? This is not a method, that I created, btw. It was contributed by other developers who were using it in there own code and thought it general enough to be reusable by others.

I think "user-code" is misleading b/c I'm not so sure it's the kind to thing one would generally use in say one's Rails model, for example (though I could well be wrong!). It's a tool geared more toward libraries. For instance one could use it very effectively for creating a DSL for database querying. The blog link I gave before is in the spirit of that kind of application. Nonetheless, I think I have an example that might fit your bill, albeit it's still abstract b/c I am not pulling it from existing code.

Consider a case of delegation in which you want to expose an object to another class, but you only want to expose a few methods, not the entire object.

```
class Example
  def thing => op, *a, &b
    @thing ||= Thing.new
    case op
    when :foo, :bar
      @thing.public_send(op, *a, &b)
    else
      raise MethodError
    end
  end
end

ex = Example.new
ex.thing.foo
```

You might think of other ways to handle this, but this approach is particularly concise and flexible (and low overhead!). Consider further if we also wanted to support another method, but with a fixed argument.

```
class Example
  def initialize(special)
    @special = special
  end

  def thing => op, *a, &b
    @thing ||= Thing.new
    case op
    when :foo, :bar
      @thing.public_send(op, *a, &b)
    when :baz
      @thing.public_send(op, @special, *a, &b)
    else
      raise MethodError
    end
  end
end

ex = Example.new("light-bulb")
ex.thing.baz
```

#13 - 07/01/2012 06:05 AM - trans (Thomas Sawyer)

- File 6594.pdf added

Here is a PDF for said proposal.

#14 - 07/02/2012 02:18 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

Received, thank you!

--
Yusuke Endoh mame@tsg.ne.jp

#15 - 07/04/2012 01:36 AM - trans (Thomas Sawyer)

FYI, I just want to make clear that the => notation I use is not vital to this proposal. It's just the first idea that occurred to me for a concise way to designate a HOM method. Another way, for example, could be hom keyword in place of def.

#16 - 10/27/2012 07:03 AM - ko1 (Koichi Sasada)

- Assignee changed from matz (Yukihiro Matsumoto) to mame (Yusuke Endoh)

mame-san, could you write the result of dev-meeting and judge this ticket?

#17 - 10/27/2012 11:13 AM - mame (Yusuke Endoh)

- Assignee changed from mame (Yusuke Endoh) to matz (Yukihiro Matsumoto)

This ticket was rejected, but the reason was too difficult for me to write.
So I asked matz to reply to this ticket.

--
Yusuke Endoh mame@tsg.ne.jp

#18 - 10/27/2012 12:03 PM - mame (Yusuke Endoh)

- Priority changed from Normal to 3

- Target version changed from 2.0.0 to 3.0

#19 - 11/27/2014 05:27 PM - nobu (Nobuyoshi Nakada)

- Description updated

#20 - 12/10/2020 08:53 AM - naruse (Yui NARUSE)

- Target version deleted (3.0)

#21 - 03/20/2024 12:21 AM - matheusrich (Matheus Richard)

@mame (Yusuke Endoh) Can we close this issue?

Files

6594.pdf	77.8 KB	07/01/2012	trans (Thomas Sawyer)