Ruby - Feature #6253

Implement a way to pass keyword options to curried procs

04/04/2012 02:27 PM - jballanc (Joshua Ballanco)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	2.0.0	

Description

=begin
(See original discussion in issue #4610)

With the introduction of keyword arguments in Ruby 2.0, it would be useful to have a way to set a keyword-based option on a curried proc. The example below demonstrates a Rack-like system where a curried proc is passed to a helper, then after partial application of the arguments, the returned curried proc is stored and later evaluated:

```
class NicknameTranslator
  def initialize(app)
    @app = app
  end

  def call(name)
    case name
    when 'Robert'
      @app.call('Bob')
    when 'James'
      @app.call('Jimmy')
    else
      @app.call(name)
    end
  end
end

class NightGreeting
  def initialize(app)
    @app = app
    @app.pass_option(greeting: 'Goodnight')
  end

  def call(name)
    @app.call(name)
  end
end

class Greeter
  def initialize(helper)
    @helper = helper
    @app = lambda do |sender, receiver, greeting: 'Hello'|
      puts "#{sender} says \"#{greeting}\" to #{receiver}"
    end.curry
    @greetings = {}
  end

  def call(sender, receiver)
    @greetings[sender] ||= @helper.new(@app).call(sender)
    @greetings[sender].call(receiver)
  end
end

Greeter.new(NicknameTranslator).call('Josh', 'Joe')
Greeter.new(NicknameTranslator).call('Robert', 'Joe')
Greeter.new(NicknameTranslator).call('Josh', 'Robert')
```

If we wanted, however, to be able to set a keyword-based option in the helper, there is currently no way in Ruby 2.0 to do so. Currently, keyword arguments can only be used at the same time as the final non-keyword, non-default, non-rest argument to the proc is applied. So, for example, there is no way to do the above with NightGreeting in place of NicknameTranslator.

Currying is really only useful when it can be used with partial application. However, Ruby currently limits how you can achieve partial application of curried procs. In particular, there is no way to manage partial application of parameters with default values. As such, it is not surprising that Proc#curry does not seem to have been adopted very widely. In my personal survey of ~600 gems that I use in various projects, I did not find any usage of Proc#curry.

So, I would request a method like Proc#pass_option (or some other, better name) that allows for setting keyword arguments on a curried proc at any time.
=end

History

#1 - 04/04/2012 09:14 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

Hello,

2012/4/4 jballanc (Joshua Ballanco) jballanc@gmail.com:

> The example below demonstrates a Rack-like system where a curried proc is passed to a helper, then after partial application of the arguments, the returned curried proc is stored and later evaluated:
> *snip*

I think Proc#curry is not an appropriate answer to your problem.
You should just create a new lambda:

```
class NightGreeting
  def initialize(app)
    @app = app
  end
  def call(sender)
    proc do |receiver|
      @app.call(sender, receiver, greeting: "Goodnight")
    end #.curry # needed when the proc accepts multiple parameters
  end
end
```

> Currying is really only useful when it can be used with partial application. However, Ruby currently limits how you can achieve partial application of curried procs. In particular, there is no way to manage partial application of parameters with default values. As such, it is not surprising that Proc#curry does not seem to have been adopted very widely. In my personal survey of ~600 gems that I use in various projects, I did not find any usage of Proc#curry.

In a sense, the situation is just as I expected :-)

I'm the person who made the proposal of Proc#curry in [ruby-dev:33676].
I suggested it with purely theoretical interest, or, just for the fun
of functional programming, such as playing with SKI combinator [1].
I did not think it would be used widely in real-world case, and even
I think it should not be used that way.

[1] http://en.wikipedia.org/wiki/SKI_combinator_calculus

Usage example in my products:

- An compiler for Grass (esoteric functional language) [2]
- An interpreter for Unlambda (yet another esoteric functional language) [3]
- An interpreter for SKI-based joke language [4]

[2] https://github.com/mame/grass-misc/blob/master/yagc.rb
[3] http://d.hatena.ne.jp/ku-ma-me/20090203/p1 (in Japanese)
[4] https://github.com/mame/ikamusume

In addition, I had another (ironical) intent; there is a lot of
confusion between curry and partial application [5].
I wanted to preempt the name "curry" in a correct terminology.

[5] http://lambda-the-ultimate.org/node/2266

--
Yusuke Endoh mame@tsg.ne.jp

#2 - 04/20/2012 04:00 AM - jballanc (Joshua Ballanco)

=begin
I still want to think about this problem some more, but I also wanted to share some of my early thoughts...

I think that part of the problem with currying in Ruby is that we don't have the sort of type system that makes curried Procs so useful in languages like OCaml. For example, if you have a function in OCaml with the signature (({A -> B -> C})), then you have a guarantee that you can apply an instance of A to the first argument, and the result will still be a function. In Ruby, if you have a method defined like (({def foo(a_proc)})), and you #call the proc passed in, you can't know whether the result will still be a proc or whether it will be the result of executing the proc. For this reason, it makes far more sense to simply wrap procs in procs:

```
def takes_curry(a_proc)
  a_proc.call('hello') # <-- might return a value, or a proc
end

def takes_proc(a_proc)
  proc do |other_arg|
    a_proc.call('hello', other_arg)
  end
  # ^ definitely returns a proc that takes one more argument
end
```

However, I don't think that means that currying procs in Ruby is useless. On the contrary, now that we have keyword arguments, I think that a curried proc could make a very compelling substitute for the method object pattern:

```
class NightParams
  def self.parameterize(meth)
    meth.greeting = 'Goodnight'
  end
end

class QuestionParams
  def self.parameterize(meth)
    meth.punctuation = '?'
  end
end

def parameterize_and_call(parameterizer, name)
  method = proc do |name, greeting: 'Hello', punctuation: '!'|
    puts "#{greeting}, #{name}#{punctuation}"
  end.curry
  parameterizer.parameterize(method)
  method.call(name)
end

parameterize_and_call(NightParams, 'moon') #=> "Goodnight, moon!"
parameterize_and_call(QuestionParams, 'Joe') #=> "Hello, Joe?"
```

Any thoughts?
=end

#3 - 06/28/2012 05:05 PM - jballanc (Joshua Ballanco)

- File 6253-proposal.pdf added

First feature request. Please let me know if PDF is not acceptable.

#4 - 07/02/2012 12:41 AM - mame (Yusuke Endoh)

Your slide is received. Thank you!

--
Yusuke Endoh mame@tsg.ne.jp

#5 - 07/02/2012 02:46 AM - trans (Thomas Sawyer)

Is the return value of #curry an instance of Method? Or is it some other thing in itself, e.g. a Curry object?

#6 - 07/02/2012 06:04 AM - jballanc (Joshua Ballanco)

=begin
Currently, Proc#curry returns an instance of Proc.

Technically speaking, currying takes an n-arity Proc, and turns it into n nested 1-arity Procs. However, Ruby's implementation does not follow this strictly. In Ruby, currying a proc prepares it for partial application, but is not strict about the 1-arity rule (and also, negative-arity on the source proc causes further confusion).

More examples (because they're fun!):

Currently -
>> def mult(a, b) a * b end
=> nil
>> times_two = method(:mult).to_proc.curry.(2)
=> #<Proc:0x007fb9341bbe90 (lambda)>
>> times_two.(4)
=> 8

Proposed future -
>> def math(a, b, function: :*) a.send(function, b) end
=> nil
>> addition = method(:math).to_proc.curry
=> #<Proc:0x007fb9341bbe90 (lambda)>
>> addition.function = :"+"
=> :+
>> add_two = addition.(2)
=> #<Proc:0x007fb9341bbe90 (lambda)>
>> add_two.(6)
=> 8
=end

#7 - 07/23/2012 11:09 PM - mame (Yusuke Endoh)

- Status changed from Assigned to Rejected

Joshua Ballanco,

Sorry but this proposal was rejected at the developer meeting (7/21).

Proc#curry is like an easter egg. Matz felt no need to care about the
interaction between Proc#curry and other advanced features.

--
Yusuke Endoh mame@tsg.ne.jp

Files

6253-proposal.pdf	30.3 KB	06/28/2012	jballanc (Joshua Ballanco)