# Ruby - Feature #14709

## Proper pattern matching

04/24/2018 03:19 PM - zverok (Victor Shepelev)

| | |
|---|---|
| **Status:** | Closed |
| **Priority:** | Normal |
| **Assignee:** | |
| **Target version:** | |

**Description**

On RubyKaigi 2017, there was a presentation of Yuki Torii about possible implementation of pattern matching.

The syntax proposed in presentation was:

```
res = [:ng, 500]
case res
when %p([:ng, status])
  p status
end
```

The proposed syntax seem to feel pretty consistent, and the implementation (forked Ruby interpreter) was working at this moment.

As @ko1 (Koichi Sasada) was one of the contributors to the experiment, I don't suppose Ruby core team is not aware of the proposal, so I'd like to know what the status of it? Are there some plans for full-strength pattern matching in Ruby 3 (with proposed, or any other, syntax)?

PS: There are many existing gems with some kind "almost real" pattern matching (including recently emerged Qo), yet I believe that the *only* useful pattern matching can be provided language core. Otherwise, two main goals can't be achieved:

- reasonable performance (as the pattern-matching is useful mostly in complicated algorithms, which tend to repeat matches thousands of times);
- unpacking of parts of the patterns into local variables.

**Related issues:**

| | |
|---|---|
| Related to Ruby - Feature #14912: Introduce pattern matching syntax | **Closed** |

### History

#### #1 - 04/24/2018 04:19 PM - shevegen (Robert A. Heiler)

I think only matz can answer that.

On a side note, I am not aware of a issue request about it. Has this already been suggested, or is it only a presentation? If it has not yet been proposed formally through an issue request on the tracker here, then perhaps that should be done.

#### #2 - 04/25/2018 12:39 AM - baweaver (Brandon Weaver)

I am the author of aforementioned Qo, so I believe I can shed some light on a few things:

- The thinking behind Qo
- Reusable syntax features
- Making it feel like Ruby
- Shortcomings of current syntax
- Performance considerations
- Ideas moving forward

I will warn you that this may be a very long post, so I will attempt to break it into the above sections to make it easier to read. The last section will propose my ideas on syntax, and will effectively summarize the others.

You can find Qo here: https://github.com/baweaver/qo

I have tried to be detailed in my documentation, and will continue to expand upon it.

## The thinking behind Qo

Qo emerged as a result of the TC39 proposal for pattern matching as well as a conversation I had at Fog City Ruby about real pattern matching in

Ruby, and made me ask a few questions:

- How can we do this using what Ruby already has?
- How can we make this look and feel like Ruby?
- How can we make the performance acceptable?

I wrote an article that explains, in more detail than I will here, the first point:
https://medium.com/@baweaver/for-want-of-pattern-matching-in-ruby-the-creation-of-qo-c3b267109b25

## How can we use current Ruby syntax and make it feel like Ruby?

Ruby is an exceptionally powerful language, and jumping back to the basics we can find === and to_proc. Both are used throughout current language features with implicit syntax:

```
# === is implied around case statements
case 'string'
when String then 1
else 2
end

# in grep
data.grep(String)

# and now in predicate methods
data.any?(String)
```

For to_proc it's a very similar story:

```
data.map(&ProcObject)
```

Both are well understood in Ruby and utilize common features. Implementing classes that expose such methods allow them to utilize these same features for more powerful APIs.

to_ary was one of my next potentials for destructuring and right hand assignment emulation in pattern matching, but I have not yet found an acceptable solution to this.

Point being, by using existing language features we retain a very Ruby-like feel:

```
case Person.new('', nil)
when Qo[age: :nil?] then 'No age provided!'
else 'nope'
end
```

Though this brings us to the next point: limitations of current syntax.

## Limitations of current syntax

One of the key items of pattern matching that I would very much enjoy is Right Hand Assignment, or the ability to inject local variables in the resulting branch of a match.

Case statements will not allow this, so something such as this is not possible as when will not be able to return a value:

```
people.map { |person|
  case person
  when Qo.m(name: :*, age: 20..99) { |person| "#{person.name} is an adult that is #{person.age} years old" }
  else Qo.m(:*)
  end
}
```

This was the reason behind me introducing the concept of match with Qo:

```
Qo.match(['Robert', 22],
  Qo.m(:*, 20..99) { |n, a| "#{n} is an adult that is #{a} years old" },
  Qo.m(:*)
)
# => "Robert is an adult that is 22 years old"

people.map(&Qo.match(
  Qo.m(:*, 20..99) { |n, a| "#{n} is an adult that is #{a} years old" },
  Qo.m(:*)
))
# => "Robert is an adult that is 22 years old"
```

It is not as elegant and Ruby-like as I would like, but it is more succinct than some of the other implementations based in more functional language paradigms using purely lambda composition.

As a result, it also ends up being faster which leads me to the next section:

## Performance Considerations

I try and be very honest about the performance concerns related to Qo, being that it is heavily dynamic in nature. Admittedly in ways it tries to be too clever, which I am attempting to reconcile for greater performance gains.

The performance results can be seen here: https://github.com/baweaver/qo/blob/master/performance_report.txt

All tests are specified in the Rakefile and compare Qo to a Vanilla variant of the same task. The performance was better than I had anticipated but still did not reach quite the speed that I had wanted it to.
I will continue to test against this and make improvements. It is likely that I can cut down this time substantially from where it currently is.

It is also the reason you may see certain === related tickets on speed improvement :)

I will endeavor to bring more comprehensive performance testing, especially around pattern matching as I have no results to this point.

## Ideas moving forward

I would be interested in a match syntax such as this:

```
new_value = match value
  when %m(:_, 20..99) { |_, age| age + 1 }
  else { |object| ... }
end
```

The block syntax will make this a less magical option as it can explicitly capture details on what was matched. Keyword arguments could be used as well, but few know of the keyword arguments in blocks:

```
json.map { |name: 'default_name', age: 42| Person.new(name, age) }
```

Such syntax exists most similarly in Scala, notably with Case Classes. Other similar implementations are present in the JS TC39 proposal and to a lesser extent in Haskell.

I have yet to think of a way to implement Left Hand Assignment like Javascript destructuring, and I do not have ideas on how to achieve such a thing beyond using to_ary for implicit destructuring.

## Conclusion

Thank you for your time in reading, and I apologize for my verbosity, but this is a feature I would very much like to see in Ruby Core some day.

I would be more than happy to provide alternative ideas on syntax in regards to this idea if you do not find the proposed ones satisfactory.

- baweaver

#### #3 - 04/25/2018 01:35 AM - baweaver (Brandon Weaver)

It was mentioned to me on Reddit that I should detail exactly what cases of pattern matching would entail.

For the case of Qo, these include four primary categories:

- Array matches Array
- Array matches Object
- Hash matches Hash
- Hash matches Object

From there each category typically descends into the following chain:

1. Wildcards: b == WILDCARD
2. Strict equality: a == b
3. Case equality: a === b
4. Predicates: a.public_send(b)

These vary between Array and Hash style matches.

This will also be long as it explains the exact contract I've implemented, so as to be a possible inspiration for design level concerns of such a feature.

## Array matches Array

An Array matched against another Array in Qo will result in them being compared positionally on intersection.

In the most basic of cases, [1,1] matched against itself would compare the elements by their index, resulting in (using a hypothetical match method):

```
match([1, 1], [1, 1]) # => true
```

```
# [1, 1]
# [1, 1]
```

My decision with Qo was to be left-side permissive on length, such that this will also return true:

```
match([1, 1], [1, 1, 2, 3, 4]) #=> true

# [1, 1]
# [1, 1, 2, 3, 4]
```

Only the first two items would be compared.

For this, comparisons will be run as:

1. Wildcard matches: b == :*
2. Strict equality: [1, 1] == [1, 1]
3. Case equality: match([Integer, 1..10], [1, 1])
4. Predicate: match([:odd?, :zero?], [1, 0])

I would say that for performance concerns case 4 could be subsumed into case 3 by transferring the symbols to Procs instead of relying on public_send.

### Array matches Object

An Array matched against an Object will attempt to compare that Object to everything in the Array via:

1. Wildcard matches: b == :*
2. Case equality: match([Integer, 1..10], 1)
3. Predicate: match([:odd?, :zero?], 1)

In some cases this can be emulated with compose as suggested in another bug with procs, but === responding values will no longer have that flexibility.

Even if an Array#to_proc were to exist, composing against === responding values would fail.

This would be more difficult to optimize for performance

### Hash matches Hash

A Hash matched against another Hash would be via intersection of keys, and in the following order:

1. Key exists: a.key?(b)
2. Wildcard matches: b == :*
3. Recurse if value is a Hash
4. Case equality: b[k] === a[k]
5. Predicate: a[k].public_send(b)
6. Repeat with String coercion of key

Qo tries to be clever here, and starts to be more strict. It wants the key to exist or it counts a non-match even if a wildcard was provided to it.

The cleverness comes into play in that it tries the string version of the key as well, just in case. This is a performance bottleneck, but also a nice to have considering how often Rubyists get the two mixed up. Problem is it adds a good deal of overhead, and keyword arguments cannot take string keys with hashrockets.

### Hash matches Object

This starts to look a lot more like ActiveRecord syntax, except in that it applies to plain ruby objects using public_send. It applies in the following order:

1. Object responds to key
2. Wildcard matches
3. Case equality
4. Predicate

It is also strict about the presence of a key / method before continuing, even if it were to be a wildcard.

Essentially each key is a method to send to the object, and each value is what it must match against somehow:

```
match(Person('Foo', 42), name: :*, age: 40..50)
```

### On Docs

A majority of this is explained in more detail in the Qo docs, as well as practical examples I've tried myself in the wild.

If you have any other questions or concerns I would be happy to address them.

#### #4 - 04/25/2018 02:53 AM - baweaver (Brandon Weaver)

It should also be mentioned that the way I achieved the pattern matching mentioned above was by using a status container in the style of Elixir ({:ok, result} except [true, result]):

https://github.com/baweaver/qo/blob/83577f80a47015e60d833da62a1220a08c00482d/lib/qo/matchers/pattern_match.rb#L77-L91

```
# Immediately invokes a PatternMatch
#
# @param target [Any]
#   Target to run against and pipe to the associated block if it
#   "matches" any of the GuardBlocks
#
# @return [Any | nil] Result of the piped block, or nil on a miss
def call(target)
  @matchers.each { |guard_block_matcher|
    did_match, match_result = guard_block_matcher.call(target)
    return match_result if did_match
  }

  nil
end
```

By inheriting from a matcher and overriding its to_proc method, one can use super to get the boolean result of the matcher. If that result is true, we can invoke the block function passed to the matcher to simulate Right Hand Assignment:

https://github.com/baweaver/qo/blob/83577f80a47015e60d833da62a1220a08c00482d/lib/qo/matchers/guard_block_matcher.rb#L38-L48

```
# Overrides the base matcher's #to_proc to wrap the value in a status
# and potentially call through to the associated block if a base
# matcher would have passed
#
# @return [Proc[Any] - (Bool, Any)]
#   (status, result) tuple
def to_proc
  Proc.new { |target|
    super[target] ? [true, @fn.call(target)] : NON_MATCH
  }
end
```

This is done to prevent false negatives from legitimate falsey values that might be returned as a result of a match, and will be a similar concern in implementation.

#### #5 - 04/25/2018 05:22 AM - baweaver (Brandon Weaver)

## On Pattern Matching in Other Languages

For continuity, we should also go over implementations in other languages.

Note that I am not an expert in any of these languages, and only proficient in Javascript. I would encourage others to source examples and correct any shortcomings of my answers.

### Javascript

I think the most practical one to read right now is the current TC39 proposal for pattern matching in Javascript:

https://github.com/tc39/proposal-pattern-matching

The pull request and discussion around it is particularly enlightening as to the challenges and concerns present in creating such a feature, and as it is a stage 0 proposal it gives a very new view to the idea as well.

An example:

```
const getLength = vector => match (vector) {
  { x, y, z } => Math.sqrt(x ** 2 + y ** 2 + z ** 2),
  { x, y } => Math.sqrt(x ** 2 + y ** 2),
  [...etc] => vector.length
}

getLength({x: 1, y: 2, z: 3}) // 3.74165
```

It would be disingenuous to not mention that Javascript also has a concept of destructuring:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

This gives it a significant advantage in defining such a pattern, and would be difficult to implement in Ruby potentially with the Hash / Block distinctions. It's possible, yes, but something to keep in mind.

Noted that I'd love destructuring as well, but that may be overkill. I wonder if it may be a prerequisite in some ways though, as Left Hand Assignment is huge to pattern matching.

## Elixir

A language directly inspired by Ruby, perhaps we can borrow back. Elixir has several types of pattern matching.

https://elixir-lang.org/getting-started/pattern-matching.html

The first should look very similar to the Javascript example, as destructuring is basically a form of pattern matching:

```
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
```

Notedly if the sides are not congruent a match will fail. This does introduce the idea that = is not really assignment, but a pattern match itself. I don't see that as an easy or really even practical thing to be able to do in Ruby.

The next is quite similar to overloading methods in Java:

https://blog.carbonfive.com/2017/10/19/pattern-matching-in-elixir-five-things-to-remember/

```
# greeter.exs
defmodule Greeter do
  def hello(:jane), do: "Hello Jane"

  def hello(name) do
    "Hello #{name}"
  end
end

# calculator.exs
defmodule Calculator do
  def multiply_by_two([]), do: return []

  def multiply_by_two([head | tail]) do
    [head * 2 | multiply_by_two(tail)]
  end
end
```

A similar idea exists in Haskell and other languages, but we'll get to them later. The concern here is that this type of technique is already done heavily in Ruby using Case as a type dispatcher:

```
def method(*args)
  case args.first
  when String then string_variant(args)
  when Integer then integer_variant(args)
 ...
end
```

Point being that pattern matching may exacerbate that issue that Elixir solves. That's also a very hard feature to account for.

## Haskell

Haskell has several different styles of pattern matching, some of which very similar to the Elixir example right above:

http://learnyouahaskell.com/syntax-in-functions

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The guard statement may be more in line with what we would recognize as a case statement:

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b
```

This also treats = as a pattern match, and relies on some of Haskell's laziness. Perhaps case expressions are the closest parallel though:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

### Scala

Scala is likely one of the closest to Ruby in terms of syntax in pattern matching:

https://docs.scala-lang.org/tour/pattern-matching.html

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}

matchTest(3)  // many
matchTest(1)  // one
```

Case classes especially are interesting in that some Ruby variants may be able to copy such techniques with constructor contracts and self.to_proc:

```
abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends Notification

def showNotification(notification: Notification): String = {
  notification match {
    case Email(email, title, _) =>
      s"You got an email from $email with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"you received a Voice Recording from $name! Click the link to hear it: $link"
  }
}

val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))  // prints You got an SMS from 12345! Message: Are you there?

println(showNotification(someVoiceRecording))
// you received a Voice Recording from Tom! Click the link to hear it: voicerecording.org/id/123
```

I could see implementations being possible off of some of Scala, but case classes aren't as useful outside the context of static types or validations. This means relying on arity or other special contracts to ensure the right thing gets matched, which could be inaccurate at times.

## Wrap Up

Noted that there are several other languages that have features such as this, but I see these as some of the more prominent ones. OCaml, F#, and other languages also feature pattern matching, and would be worth looking into.

Most of Qo's implementation is loosely based off of Scala and Haskell techniques.

#### #6 - 04/25/2018 07:45 PM - baweaver (Brandon Weaver)

On considering a bit more, I've come up with a few syntax variants for such a feature.

Preference towards avoiding %p and other shorthands in favor of a more self-descriptive word like matches or match.

### 1. matches

There are a few variants of this I can think of.

#### 1A - Matches Guard Block

The first involves the guard-block style syntax:

```
case value
```

```
when matches(/Foo/, 42) { |name, age| }
else matches { |object| }
end
```

**1B - Matches then local inject**

The second would involve the use of then or indent:

```
case value

# First option is to use then as a demarcation like if and case/when:
when matches(/Foo/, 42) then Person.new(name, age)

# alternatively, line break:
when matches(/Foo/, 42)
  Person.new(name, age)

else matches
  # alternate case
end
```

Considerations for this second method are how you would define local variables. Blocks give you control over this, but Ruby does not currently have a clean way to define local variables based off the results of a function.

With something like case classes it'd be a matter of reflection on arity and param names from the initializer, but this would be slow. It'd also be potentially rife with namespace collisions and shadowing by trying to be too clever.

## 2. Enumerator-like yielder:

Enumerator has the following syntax which exposes a bind point:

```
one_then_two_forever = Enumerator.new do |y|
  y.yield(1)
  loop { y.yield(2) }
end
```

What if we extrapolate from that?:

```
match(value) do |m|
  m.when(/name/, 42) { |name, age| Person.new(name, age) }
  m.else { |object| raise "Can't convert!" }
end
```

This would be similar to concepts used for benchmark, websockets, async, and other tasks by providing a yielder point. It is also very succinct.

It may be a bit magic, but when passed no argument match could return a Proc instead awaiting a value. That would allow for some really interesting things like this:

```
def get_url(url)
  Net::HTTP.get_response(URI(url)).then(&match do |m|
    m.when(Net::HTTPSuccess) { |response| response.body.size }
    m.else { |response| raise response.message }
  ))
end
```

Any one of the when cases could be used with ===, behaving very much like a case statement's when. Right-hand destructuring may still be interesting here though, I'll have to think on that.

This syntax would expose a very minimal changeset to the language itself, requiring only a single word and utilizing already common patterns.

**EDIT** - I've just finished an implementation of that variant, and the syntax for the two above examples using it would be:

```
Qo.match(value) do |m|
  m.when(/name/, 42) { |name, age| Person.new(name, age) }
  m.else { |object| raise "Can't convert!" }
end

def get_url(url)
  Net::HTTP.get_response(URI(url)).then(&Qo.match do |m|
    m.when(Net::HTTPSuccess) { |response| response.body.size }
    m.else { |response| raise response.message }
  ))
end
```

So it would appear to be very minimally invasive as far as syntax additions. The code for it is in lib/qo/matchers/pattern_match_block.rb, and I've

commented on the differences there.

#### #7 - 04/27/2018 02:01 AM - jgaskins (Jamie Gaskins)

Whenever I feel I need pattern matching, I'm trying to handle method args. With some of the ideas presented here, like Qo, I feel I'd really only be using it inside the outer edges of a method to match on the args that that method received — that is, I'd have to manually pipe the args I received into the pattern matcher every time. I automated that concept in one of my apps, from which I extracted a gem called method_pattern (it very slightly predates Qo). The idea behind it, following baweaver's Net::HTTP example:

```
class Client
  extend MethodPattern

  defn :get do
    with(String) { |url| request(URI(url)) }
    with(URI) { |uri| handle_response Net::HTTP.get_response(uri) }
  end

  defn :handle_response do
    with(Net::HTTPOK) { |response| JSON.parse(response.body) }
    with(Net::HTTPNotFound) { raise ArgumentError, "URL doesn't point to a valid resource" }
  end
end
```

This example is only based on classes, but it also works for values because it's based on the === method, as well — it really is such a great tool for pattern matching. Another example here, for a method def get(status:, headers:, body:), which also shows how handling keyword args can look (even when matching only a subset of them), and even matching nested hashes as in Yuki Torii's pmruby:

```
class Client
  extend MethodPattern

  defn :get do
    with(status: 200...400, headers: { 'Content-Type' => /json/ }) do |body:, **|
      handle_success JSON.parse(body, symbolize_names: true)
    end

    with(status: 400...500, headers: { 'Content-Type' => /json/ }) do |body:, **|
      handle_client_error JSON.parse(body, symbolize_names: true)
    end

    with(status: 500..599) do |body:, **|
      handle_server_error body
    end
  end
end
```

I don't know what first-class syntax could look like for this, unfortunately, but I imagine it might look similar to whatever ideas people had for type annotations. The hard part is making it work with multiple entries per method.

#### #8 - 04/27/2018 08:25 AM - baweaver (Brandon Weaver)

I've gone and done something exceptionally bad to Ruby, but this should be an interesting proof of concept nonetheless:

- Benchmarks - https://github.com/baweaver/qo/blob/evil/Rakefile
- Results - https://gist.github.com/baweaver/0869255d7325000c4c9bca5c836aef3c

Qo::Evil runs within decent speed parity of vanilla Ruby on arrays of any significant size (need to qualify, but around 100+ I'd guess), and faster than any dynamic style vanilla Ruby.

The current branch is super messy, but it's an interesting idea:

https://github.com/baweaver/qo/blob/evil/lib/qo/evil/matcher.rb

Essentially take a list of matchers and "compile" them into a proc using eval and some caching against a list of safe values, binding any non-coercibles to a local variable stack that gets set to the binder.

Be aware, this is very much black magic, but it's black magic that very much amuses me as it means something quite fun: It's possible to make pattern matches run at close (~1.2 - 1.5x) to optimal vanilla Ruby speed without even touching C code.

#### #9 - 06/21/2018 08:36 AM - matz (Yukihiro Matsumoto)

*- Status changed from Open to Closed*

We are not going to add the pattern matcher proposed in the OP (that uses %p), because it is a mere prototype. Yuki told us so clearly. If we were going to add pattern matching in Ruby, we should add it with better syntax.

Regarding Qo, I like the basic idea, but similarly, we should consider new (and good looking) syntax.
The problem is that I have no idea for an excellent syntax for the pattern matching right now.
I close this issue and expect the proposal for a new syntax.

Matz.

#### #10 - 06/27/2018 08:18 AM - shevegen (Robert A. Heiler)

zverok wrote an add-on in regards to "syntax proposal" here:

https://zverok.github.io/blog/2018-06-26-pattern-matching.html

I don't have a suggestion for a better syntax myself really.

zverok suggested, among other examples, e. g. this:

```
case (lat, *lng)
when (Numeric, Numeric => lng, Hash => opts)
```

I am not a huge fan of the (). :P

However had, I was also thinking ... pattern matching is in
some ways similar to regex-checking, yes? For regexes we
have =~.

Perhaps we could use a similar construct for pattern matching,
also starting with =. Not sure about the second character.

=| might be almost an obvious choice since it looks like a
pipe in a way, but I am not sure if this can be used.

One could also ponder about a combination of three characters,
a bit like the somewhat new one used for lambdas. ->| or
->~  ... not that any of these are very pretty. And one thing
I also always mentioned is that it should not be too hard to
visually differentiate between different things in ruby code
so perhaps something like ->~ is a bad idea since it is not
so easy to see any difference. But two characters may be fine,
like the regex variant or also the compound ones like += or
-= (two characters seem better than more than two characters).

#### #11 - 08/01/2018 02:32 PM - ktsj (Kazuki Tsujimoto)

*- Related to Feature #14912: Introduce pattern matching syntax added*