Ruby - Feature #18773

deconstruct to receive a range

05/11/2022 05:13 PM - kddnewton (Kevin Newton)

Status:	Rejected
Priority:	Normal
Assignee:	ktsj (Kazuki Tsujimoto)
Target version:	

Description

Currently when you're pattern matching against a hash pattern, deconstruct_keys receives the keys that are being matched. This is really useful for computing expensive hashes.

However, when you're pattern matching against an array pattern, you don't receive any information. So if the array is expensive to compute (for instance loading an array of database records), you have no way to bail out. It would be useful to receive a range signifying how many records the pattern is specifying. It would be used like the following:

```
class ActiveRecord::Relation
  def deconstruct(range)
    (loaded? || range.cover?(count)) ? records : nil
  end
end
```

It needs to be a range and not just a number to handle cases where * is used. You would use it like:

```
case Person.all
in []
  "No records"
in [person]
  "Only #{person.name}"
else
  "Multiple people"
end
```

In this way, you wouldn't have to load the whole thing into memory to check if it pattern matched. The patch is here: https://github.com/ruby/ruby/pull/5905.

History

#1 - 05/12/2022 07:54 PM - kddnewton (Kevin Newton)

- Description updated

#2 - 06/14/2022 04:34 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to ktsj (Kazuki Tsujimoto)

#3 - 06/14/2022 07:23 AM - mame (Yusuke Endoh)

It would be easier to discuss if you could write a spec of what pattern match will pass what range. I understand as follows by reading your implementation. Right?

- ary in [1, 2, 3] will call ary.deconstruct(3..3), which means "the length must be exactly 3"
- ary in [1, 2, *, 3] will call ary.deconstruct(3..), which means "the length must be greater than or equal to 3"

I understand your motivation, but I wonder if the spec could be more efficient. In the second calling sequence, the match requires only the first, second and last elements, but ary.deconstruct(3..) needs to create an array including all elements because it does not know which elements are required.

Though the current implementation of pattern matching is not so efficient, but I am afraid that the proposed implementation looks very inefficient because it creates a Method object and calls #arity.

#4 - 06/24/2022 07:21 AM - ktsj (Kazuki Tsujimoto)

As a designer of pattern matching, I also understand your motivation.
However, I have the following concerns in addition to the ones mame pointed out.

- In the current implementation, when a pattern match fails in one-line pattern matching, the reason for the failure is displayed as an error message. I think it difficult to do the same thing by your proposal. (This feature is not a must, but it would be nice to have.)

```
$ ruby -e '[0] => []'
-e:1:in `<main>': [0]: [0] length mismatch (given 1, expected 0) (NoMatchingPatternError)
```

- It might have a negative impact on performance. The current implementation caches the return value of deconstruct method, but this approach will no longer be available.

#5 - 07/03/2022 06:04 PM - kddnewton (Kevin Newton)

@mame (Yusuke Endoh) and @ktsj (Kazuki Tsujimoto) - I definitely understand your concerns! I'm sorry I didn't make it clear in the initial report, I meant to open this ticket as the beginning of a discussion, not as the definitive solution. I would be very open to discussing other options.

Mostly I'm just trying to solve the issue where loading an array with deconstruct is expensive (like deconstruct_keys with the keys argument). I can think of a couple of ways to do this, including the approach in this commit:

- Deprecate deconstruct without an argument, eventually pass it a range that functions as @mame (Yusuke Endoh) described
- Deprecate deconstruct without an argument, eventually pass it a size for before, a boolean for *, a size for after
- Create a new method deconstruct_indices that receives those arguments, call it if it is defined, otherwise call deconstruct if it is defined
- Do nothing and not worry about it □□

I think any of these would be okay. I agree with @mame (Yusuke Endoh) that creating a method object is probably a bad idea.

#6 - 07/04/2022 03:07 AM - nevans (Nicholas Evans)

@kddeisz If we want that second option to short-circuit on min size, we need to send min independently of max_potential_pre_args and max_potential_post_args. E.g. we wouldn't want to accidentally return nil for this *(edited for clarity)*:

```
# min array pattern args: 5
# min find pattern  args: 1
case obj
in [A,B,C,D,*,Z];   etc
in [A,B,C,*,Y,Z];   etc
in [*, {find:}, *]; etc
end
```

And if we want to return the smallest array containing all potentially used args, [pre, *, post] can return *only* pre and post elements... with a caveat: disambiguating between when the max args comes from a pattern without a rest-arg, but another pattern has a rest arg. E.g. case obj; in [A] then ...; in [a, *] then ... end. In that case, we also need at least one padding value. It will be completely ignored other than its affect on the array's size, so it can be anything, e.g. nil or :deconstruct_padding. We can request that the padding is *always* added, or we can send a bool when we really need it. I'd favor sending the bool, just to *remind* people it's needed. I know I'd forget one day.

So I think the following signature would be sufficient for both short-circuiting and fetching everything necessary into the smallest possible array:

```
def deconstruct: () -> Array # no min, no max, return everything
               # must return all, but may short-circuit if size<min || max<size
               | (Integer min, Integer? max) -> Array?
               # may return array with only pre and post, w/o rest (maybe +1)
               | (Integer min, boolean pad, Integer pre, Integer post) -> Array?
```

And some examples (For simplicity and terseness, let's assume the following alternative patterns could also be used as multiple "case in" clauses. So, e.g. order and identifying the "winner" matters.):

EDITED the following examples - because it's confusing and I got them wrong the first time...

```
# Padding is unnecessary for disambiguation when:
#  * only one alternative,
#  * or only pre and post args (no rest),
#  * or contains alternatives with both pre and post args,
#  * or contains an unbound rest-arg,
#  * or largest args w/o rest < max pre + max post
obj.deconstruct(3, false, 2, 1) in [1, 2, *, 3]
obj.deconstruct(2, false, 3, 0) in [A, B, C, *] | [a, b]
obj.deconstruct(6, false, 6, 3) in [A, B, C,  *,  X, Y, Z] | [a,b,c,x,y,z]
obj.deconstruct(3, false, 6, 3) in [A, B, C,*]|[*,X, Y, Z] | [a,b,c,x,y,z]
obj.deconstruct(4, false, 4, 4) in [A, B, C, D, *] |
                                   [A, B, C, *, Z] |
                                   [A, B, *, Y, Z] |
                                   [A, *, X, Y, Z] |
                                   [*, W, X, Y, Z] |
                                   [a, b, c, y, z]

# Padding is needed for disambiguation when:
#  * multiple alternatives,
```

```
#  * and at least one unbound rest-arg,
#  * and no bound rest-args (which would need to fetch everything),
#  * and only pre-args or only post-args (not both),
#  * and largest max pre/post args w/ rest <= max args w/o rest
obj.deconstruct(0, true,  1, 0) in [] | [person] | [person, *]
obj.deconstruct(1, true,  3, 0) in [a, b, c] | [A, *]

# Always handling "no rest arg" as pre-arg:
obj.deconstruct(3, false, 3, 3) in [A, B, C] | [*, X, Y, Z]
# Consolidating "no rest arg" with post-arg when no pre-args are used:
obj.deconstruct(3, true,  0, 3) in [A, B, C] | [*, X, Y, Z]
```

I've actually been thinking about this ticket on-and-off for the last week or so...
I wrote a *much* larger response, and I've been struggling to edit it to something reasonable! :)
Anyway, I came up with a different approach for you to consider, which I'll post next. I hope you'll find it simple, extensible, and very ruby-ish. :)

#7 - 07/04/2022 03:21 AM - nevans (Nicholas Evans)

So here's another option: allow deconstruct to return an Enumerable (or duck-typed). Personally, I'd *much* rather write custom optimized versions of enumerable methods than write a complicated optimized deconstruct method. And if the methods to efficiently pattern match don't currently exist on Enumerable, perhaps they should be added? That way improvements to Enumerable can automatically become improvements to pattern matching and vice versa.

I was thinking something along the lines of:

- v1: short-circuit using enum.size then proceed with enum.to_a as the input
- v2: fill input array with only values that can potentially be used: [*enum.first(pre.max), *enum.last(post.max)]
  (This might require Enumerable#last. But we already have #reverse_each, and other O(n) or O(n lg n) methods, so why not #last?)
- v3: use the enumerator directly (much more complicated than v1 and v2). enables lazy-loaded short-circuiting.
  - enables find patterns to match against lazily fetched elements, and lazy loading for potential *but unlikely* pre & post args
  - might require a caching proxy enumerator, or Enumerator#next,
    or a new method like split_at(idx) => [Enumerable => head, Enumerable => tail]
- v4: handle unbounded/infinite sequences with *lazy-loaded rest vars*. This is maybe a crazy thing, but I think it could be very nice: allow stream in [a, b, *rest] to assign an *enumerable* to rest, which can then be used as a continuation.

#8 - 07/04/2022 05:38 PM - nevans (Nicholas Evans)

Oops, I made some mistakes in my "needs disambiguation" examples. Assuming that pre+post members will always be returned when present, padding is only needed when the largest pattern with no rest arg is equal to pre+post. Which means it isn't necessary if both pre *and* post exist in any clauses. So:

```
# needs padding: 6 == 6+0
#     deconstruct(3, true, 6, 0)
case obj
in a,b,c,x,y,z
  # args:6, rest:false
in A,B,C,*
  # pre: 3, rest:true
in A,B,c,d,*
  # pre: 4, rest:true
end

# only post args: depends on whether we consolidate no-rest args with post-args,
# or simply always count them as pre-args
# does need padding:  6 == 0+6
#     deconstruct(3, true, 0, 6)
# padding not needed: 6 <  6+4
#     deconstruct(3, false, 6, 4)
case obj
in a,b,c,x,y,z
  # args:6, rest:false
in *,    X,Y,Z
  # post:3, rest:true
in *,  W,X,Y,Z
  # post:4, rest:true
end

# doesn't need padding.
#     deconstruct(3, false, 6, 3)
case obj
in a,b,c,x,y,z
  # args:6, rest:false
in A,B,C,*
  # pre: 3, rest:true
in *,X,Y,Z
```

```
  # post:3, rest:true
end
```

Of course, it would be fine to *always* send padding whether needed or not.

But it would be simpler to lean on Enumerable. We don't need deconstruct to short-circuit based on size and slice the data; that duplicates logic that's already in the pattern match code. Creating the input array using size, to_a, first, and last gives us everything that deconstruct could.

And theoretically, if/when pattern matching can lazily iterate, an enumerator could be instrumented and feed that into a query optimizer, which might know which args are unlikely and which should be fetched eagerly.

e.g. no use fetching rest and post args if we only rarely need them:

```
case data_scope
in Likely, ToSucceed, *
  # only need first 2
in Unlikely, ToSucceed, *rest, Tail
  # fetch tail? fetch everything?
in Rarity, a,b,c,d,e,f,h,i,j,k,l,m
  # maybe don't prefetch all 13?
end
```

#9 - 07/22/2022 05:48 PM - kddnewton (Kevin Newton)

@nevans I think that's a *really* good idea. It has the advantage of being backward-compatible as well. @mame (Yusuke Endoh), @ktsj (Kazuki Tsujimoto) what do you think about allowing Enumerable to be returned from deconstruct and using that to match instead?

#10 - 06/07/2024 01:51 PM - kddnewton (Kevin Newton)

- Status changed from Assigned to Rejected

Going to close this, as I think it should be fixed in other ways.

#11 - 06/07/2024 03:33 PM - Eregon (Benoit Daloze)

The implementation of pattern matching needs to read at indices from the return value of deconstruct, but Enumerable doesn't have [] or at or anything like that.
So I think it is difficult to support and could very well end up making pattern matching slower.

There is a similar issue in the issue description, it would prevent caching the result of deconstruct, as already pointed out.

I think pattern shouldn't do too crazy things (like calling each with a block behind the scenes), I think simple is good.

#12 - 06/07/2024 03:36 PM - Eregon (Benoit Daloze)

FWIW I think CRuby currently support subclassing Array and redefining [] and pattern matching will call that on CRuby, maybe it can be used for this purpose.
I don't think it's a good idea to rely on this behavior though, it might change. For example that doesn't work on TruffleRuby, which currently always uses standard Array#[] from pattern matching.