Spreading empty arrays works, even when they go through a variable, or are disguised:

```ruby
args = []
# => []
->{}.call *[] # => nil
->{}.call *args # => nil
->{}.call *([]) # => nil
->{}.call *([];[]) # => nil
->{}.call *([[]]) # => nil
->{}.call *[args] # => nil
```

Spreading empty keywords does not, when going through a variable, or sufficiently disguised:

```ruby
kws = {}
# => {}
->{}.call **{} # => nil
->{}.call **kws rescue $!
# => #<ArgumentError: wrong number of arguments (given 1, expected 0)>
->{}.call **{} # => nil
->{}.call **{} # => nil
->{}.call *{} rescue $!
# => #<ArgumentError: wrong number of arguments (given 1, expected 0)>
->{}.call **{} # => nil
->{}.call **{} rescue $!
# => #<ArgumentError: wrong number of arguments (given 1, expected 0)>
->{}.call **{**kws} rescue $!
# => #<ArgumentError: wrong number of arguments (given 1, expected 0)>
```

It seems that **{} gets optimized out of the code, as expected. Likely due to https://bugs.ruby-lang.org/issues/10719
But **empty_kws still gets incorrectly passed as a hash, despite an attempt to fix it in https://bugs.ruby-lang.org/issues/13717

```ruby
->a{}.call **{} rescue $!
# => #<ArgumentError: wrong number of arguments (given 0, expected 1)>
->a{}.call **kws # => {}
->a{}.call **{v} # => {}
{v} # => {}
```

Further confusion, it's missing a, not b:

```ruby
->a,b:{}.call **{b:1} rescue $! # => #<ArgumentError: missing keyword: b>
```

Treating keywords as a special form of hash makes them very difficult to reason about. Arrays manage to pull off destructuring and spreading with no issue, as we saw above. I just want hashes to work like arrays with named matching instead of ordinal matching.

For each example below, try looking at the LHS and predicting what the result will be.

```ruby
->a,b:,**c([a,b,c]).call 1, b:2 # => [1, 2, {}]
->a,b:,**c([a,b,c]).call 1, b:2, 3=>4 rescue $!
# => #<ArgumentError: wrong number of arguments (given 2, expected 1; required keyword: b)>
->a,b:,**c([a,b,c]).call 1=>2, b:3 rescue $! # => #<ArgumentError: missing keyword: b>
```
Keywords are getting in the way of beautiful hash spreading!

```ruby
> a, b,**c([a, b, c]).call 1=>2, **(b:3) rescue $!  # => #<ArgumentError: missing keyword: b>
> a, b,**c([a, b, c]).call 1=>2, b: 3  # => [{1=>2}, 3, {b:3}]
> a, b,**c([a, b, c]).call 1=>2, (b: 3)  # => [{1=>2}, 3, {b:3}]
> a, b,**c([a, b, c]).call 1, b:2, c:3, 4=>5  # => [1, (:b=>2, :c=>3, 4=>5)]
> a, b,**c([a, b, c]).call 1, b:2, c:3, 4=>5  # => [{1, (4=>5)}, 2, {:c=>3}]
```

$$node -v$$
```
# >> v8.9.4
```

$$node -p '({a, c, ...rest}) => [a, c, rest])
  ({a: 1, b: 2, c: 3, d: 4})
',
# >> [ 1, 3, { b: 2, d: 4 } ]
```

```bash
$ node -p '{...{a: b}, ...{c: d}, ...e}
  ;
',
# >> { a: 1, b: 2, c: 3, d: 4, f: 5, g: 6 }
```

**Related issues:**

**Related to Ruby master - Feature #14183: "Real" keyword argument**

**Closed**

### History

**#1 - 01/29/2018 03:04 AM - josh.cheek (Josh Cheek)**

Was thinking about this more, and I **think** I see what the problem is: **`** should not be kwrest, it should be options_rest. And keyword args should be about destructuring the options hash. In the case of mixed keys in the hash, they are valid options to pass to **var, even though they cannot be destructured. Here are some examples:

```ruby
# This behaves correctly: if you destruct the options hash,
# than anything not accounted for should explode
-> a='a', b:'b' { [a, b] }.call a: 2, b: 3 rescue $!
# => #<ArgumentError: unknown keyword: a>

# This should have done what the previous example did.
# Instead, it pulls `(1=>2)` out and assigns it to `a`
-> a='a', b:'b' { [a, b] }.call 1 => 2, b: 3
# => [{1=>2}, 3]

# This is the same as the previous example, but without the `(1=>2)`
# it behaves correctly.
-> a='a', b:'b' { [a, b] }.call b: 3
# => ["a", 3]

# Here, the parameters make no use of keywords, so we correctly
# treat it like a hash, from the Ruby of old.
-> a ( a ).call b: 3
# => [:b=>3]

# This should be an argument error, the method receives an options hash,
# an options hash was passed, so `b:3` should be assigned to `b`, and
# the missing argument `a` should cause an explosion.
-> a, **b { [a, b] }.call b: 3
# => [{:b=>3}, {a}]
```

**#2 - 06/20/2019 08:28 PM - jeremyevans0 (Jeremy Evans)**

- Related to Feature #14183: "Real" keyword argument added
With recent changes to the master branch, you now get the following results:

```ruby
kws = {}  
->[].call **kws  # => nil  
->[].call *[*]  # => nil  
->[].call **({})  # => nil  
->[].call **([])  # => nil  
->[].call **(kws)  # => nil

->a[].call **() rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>
->a[].call **(b) rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>
->a[].call **(); rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>
->a[].call **();(b) rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>

->a[a].call *(b) rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>
->a[a].call **(kws) rescue $! # => #<ArgumentError: wrong number of arguments (given 0, expected 1)>

->a[a].call *(b:1) rescue $!  
# warning: The keyword argument for `call' (defined at (irb):13) is passed as the last hash parameter  
# => #<ArgumentError: missing keyword: :b>

->a[b][a].call *(b:2)  
# => [1, 2, [{}]]
->a[b][a].call *(b:2, 3=>4)  
# => [1, 2, [{3=>4}]]
->a[b][a].call *(l=>2), b: 3)  
# => [{l=>2}, 3, [{}]]

->a[a][a].call *(b:c,d)  
# => [1, 2, :c, :d]
->a[a][a].call *(b:c,d)  
# => [1=>2, :c=>3, 4=>5]

->a[a][a].call *(b:).call (l=>2), b: 3)  
# => [1=>2, 3, [{}]]

->a[] a].call 1, b:2, c:3, 4=>5  
# => [1, {b=>2, :c=>3, 4=>5}]
->a[b][a].call *(b:1).call (l=>2), b: 3)  
# => [{l=>2}, 3, [{}]]

[*[1,2], *[c, d])  
# => [1, 2, :c, :d]
[**[1=>2], *[c, d, :a:3])  
# => [1, 2, [:a=>3]]
[1,2,**[a:3])  
# => [1, 2]
[1,2,**kws])  
# => [1, 2, [{}]]

-> a=a', b: 'b' { [a, b] }.call a: 2, b: 3 rescue $! # => #<ArgumentError: unknown keyword: :a>

-> a=a', b: 'b' { [a, b] }.call 1 => 2, b: 3
# warning: The last argument for `call' (defined at (irb):2) is split into positional and keyword parameters 
# => [[1=>2], 3]

-> a=a', b: 'b' { [a, b] }.call b: 3 => [*a, 3]

-> a ( a ).call b: 3
# => (b=>3)

-> a, **b { [a, b] }.call b: 3 rescue $!  
# warning: The keyword argument for `call' (defined at (irb):5) is passed as the last hash parameter  
# => [#{b=>3}, [{}]]
```

For the calls that warn, you will get the following behavior in Ruby 3:

```ruby
->a[b][a].call **(b:1) rescue $!  
# => ArgumentError (wrong number of arguments (given 0, expected 1))

->a[b][a].call *(a,b,c).call l=>2, b:3 rescue $!  
# => ArgumentError (wrong number of arguments (given 0, expected 1))

->a[b][a].call *(a,b,c).call l=>2, **(b:3)
# => ArgumentError (wrong number of arguments (given 0, expected 1))

->a[b][a].call *(a,b,c).call (l=>2), (b: 3)
# => ArgumentError (wrong number of arguments (given 2, expected 1))
```
-> a='a', b='b' { [a, b] }.call l => 2, b: 3
  # [a', [l => 2, :b => 3]]

-> a, **b { [a, b] }.call b: 3 rescue $!
  # => ArgumentError (wrong number of arguments (given 0, expected 1))

I think the only case that is questionable still is:

```ruby
[**({});]  # => []
```

The keyword argument separation changes just made to the master branch did not affect this code, since it isn't a method call. This behavior has been present since Ruby 2.2. I think it would be a good idea to make both ```[**{};]``` and ```[**{}]``` return `[]`.

**Found some other interesting cases which I absolutely don't understand:**

```ruby
[**({};)]  # => []
```

But I must admit I don't really see the benefit to all this, since it only works on hash literals and not hash variables. In what circumstance is it helpful to have a double-splatted empty hash literal in an array???

Recent changes to the master branch have fixed this issue. Keyword splats of empty hashes in arrays no longer add an empty hash to the array:

```ruby
[**{}];]  # => []
```