Why are many Clojure functions variadic?

Asked 13 years, 2 months ago Modified 13 years, 2 months ago Viewed 3k times

14

Here's an issue I keep running into in Clojure:

```
user=> (max [3 4 5 6 7])
[3 4 5 6 7] ; expected '7'
```

Some functions don't do what I expect!
Here's one solution using `apply`:

```
user=> (apply max [3 4 5 6 7])
7
```

Other examples are `concat`, and `min`.
My question, as a Clojure newbie, is why are these functions variadic? I expected them to operate on sequences. Is using `apply` the best/idiomatic way to get what I want?

Note: I'm not trying to say that it's bad to have variadic functions, or that there is a better way. I just want to know if there's a rule or convention being followed, or if there are specific advantages to such an approach that I should be aware of.

Edit: I think the original question was unclear. Here's what I meant:

In other programming languages I've used, there are monoid-like operations, such as adding numbers, finding the greater element, concatenating lists.

There are often two use cases for these operations:

1) combining two elements, using a function that accepts two arguments
2) combining 0 to n elements, using a function that accepts a list (or sequence) of elements

A function for the second case can be built from that for the first case (often using `reduce`).

However, Clojure adds a third use case:

3) combining 0 to n elements, using a variadic function

So the question is, why does Clojure add this third case?
Paul's answer indicates that:

- this allows code that is more flexible
- there are historical forces at work

clojure variadic-functions

Share
Improve this question
Follow

edited Oct 25, 2011 at 13:41

asked Oct 19, 2011 at 15:10

Matt Fenwick
49.1k ● 24 ● 129 ● 198

Personally I think this is a great question and something I still find consternation in day to day use of Clojure. The whole point of using Clojure (to me) is to build everything out of one simple and consistent yet infinitely powerful set of building blocks. Variadic functions compromise what would otherwise be maximal consistency between all of the various parts, basically just disguising a seq as a seamless part of the parameters. So, really you are just passing a seq, but now there are two slightly different ways to pass a seq. Gah! Such a blight on an otherwise beautiful language. – prismofeverything Apr 3, 2017 at 0:22

2 Answers

Sorted by: Highest score (default)

17

1) **Convenience.** With math functions such as `+` it would be annoying to wrap everything in sequences when you just try to do some calculations.

2) **Efficiency.** Wrapping everything with collections/sequences would also be inefficient, first the sequence needs to be created and then it needs to unpacked at runtime instead of looking up the right Java function at compile time.

3) **Expectations.** This is how these functions work in other Lisps and similarly, with a somewhat different syntax, in other functional languages, so it is a reasonable expectation for people coming to Clojure. I would say that the idiomatic way in other functional languages to apply a function such as `+` to a sequence would be to use either `reduce` or `foldl` / `foldr`, so this is also in line with how Clojure handles it.

4) **Flexibility.** The fact that these functions may be used with higher order functions, such as `map` makes them more convenient to use if they are variadic. Say you have three vectors and you want to apply a function to the elements at the same position. If your function is variadic then you can just use map with multiple collections (map also has to be variadic then ;)):

```
(map + [1 2 3 4] [2 3 4 5] [3 4 5 6])
; [6 9 12 15]
```

This is a lot more convenient than what you would have if all those functions just took collections.

Idiomatic use: (Edited after kotarak's excellent comment)

It depends on the function if you should use `reduce` or `apply`.

For math functions (`+,-,*,/,etc.`) that take 2 arguments in the Java world `reduce` makes more sense as it can directly use the 2 argument Java version. With `apply` they kind of do an implicit `reduce` (the function adds two arguments and then recurs with the result, the next argument and the rest. That's pretty much what reduce does.)

For `str` using `apply` is likely more efficient. When `str` is called with more than one argument it creates a StringBuilder, adds all the arguments to it and then creates a string. Using `reduce` the StringBuilder will be created n-1 times and only add one string each time. This is as in the Shlemiel the painter joke, resulting in O(n^2) complexity.

Verdict so far: Using `apply` with math functions doesn't hurt much, but using `reduce` with `str` may be pretty expensive.

Share

Improve this answer

Follow

edited Oct 21, 2011 at 8:00

answered Oct 19, 2011 at 15:17

Paul

8,132 ● 3 ● 43 ● 50

1 I disagree with 1 and 2: it's inconvenient and either less efficient or the same for me, since I'm starting with sequences. `+` is a red herring, because sometimes it acts like binary addition, and sometimes like a `sum` function -- I was discussing the `sum` part of its behavior; I'll remove it from the OP. But good points for 3 and 4, and nice example! – Matt Fenwick Oct 19, 2011 at 15:38

1 1 and 2: You may start with a sequence, but not everybody does! I suppose these functions *could* be overloaded for sequences now that clojure has deftype/defrecord and protocols, I am not sure if that was possible entirely in Clojure before that. If you don't have type-dependent dispatch and you look up how to dispatch (either on numbers or a sequence) at runtime you are paying a performance penalty. – Paul Oct 19, 2011 at 15:51

1 I don't think there are, at least not in clojure.core. But I think `(reduce + [yourlist])` and similarly for `max` and so on may actually be shorter or at least not much longer than writing your own. I think that is probably also why they don't exist yet. – Paul Oct 19, 2011 at 16:01

Don't roll sum-off-list etc functions. Apply is the idiomatic way to use a collection as parameters for a function. – NielsK Oct 20, 2011 at 15:37

3 Whether to use `reduce` or `apply` depends on the function. For `+` it's `reduce` because the two argument version can be inlined, while higher variadic version can not. They do a

`reduce` under the hood anyway. For `str` it's `apply` because `str` can then re-use the underlying `StringBuilder`. So it is more efficient and generates less garbage of intermediate strings as it would with `reduce`. So neither `reduce` nor `apply` is more idiomatic than the other. – kotarak Oct 21, 2011 at 7:31

Alex Miller just wrote about a related issue in his blog: 2 is a smell

3

Share Improve this answer Follow

answered Oct 19, 2011 at 16:22

Julien Chastang

17.8k ● 12 ● 65 ● 89