Extracted from:

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code

This PDF file contains pages extracted from *Mastering Clojure Macros*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2014 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

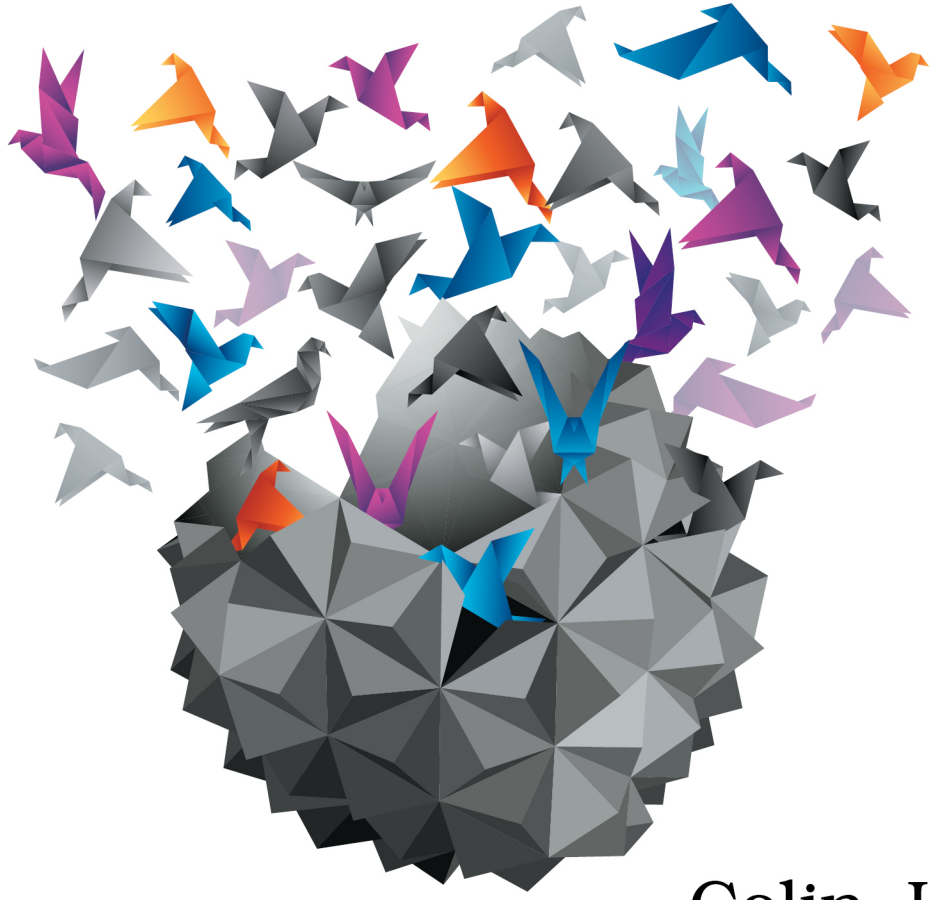
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code



Colin Jones

Edited by Fahmida Y. Rashid

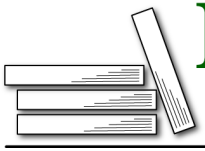
Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code

Colin Jones

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Fahmida Rashid (editor)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2014 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-941222-22-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2014

So there you are, deep in the midst of a task to analyze your website traffic logs. You were hoping to have finished it yesterday, so your heart leaps as you realize you finally have everything tested and working. Now it's time to kick off the task on the full dataset, and wait. And wait. And wait. How long will it take to finish? A day? A week? Until the heat death of the universe? After about an hour, things are looking pretty bleak.

Now, Clojure is fast. One of the language's guiding principles is that it should be useful wherever Java is useful, including use cases that require high performance, like your log analysis work. As with any other programming language, there are times when we'll need a bag of tricks to make our code as fast as possible. Sometimes, performance hacks like type hints and other Java interop artifacts result in uglier code. But with macros, we can hide the noise and maintain the beauty of our code. You'll see how in this chapter, and you'll also see how you can *entirely avoid* runtime costs in some situations with macros. But first you need to make sure you optimize the right things by benchmarking.

Benchmarking Your Code

No discussion about performance would be complete (or to my way of thinking, even worth beginning) without that famous quote from *Donald Knuth [Knu74]*:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

If we apply our time-limited efforts randomly around the codebase, it may not improve any bottlenecks that exist. My experience tends to align with Knuth's advice here: it saves us time, and a lot of wasted effort, to first find problem areas in our programs. If our program doesn't need to be fast, *any* performance work is unnecessary, or at least, premature. If our web application spends the majority of its time on I/O tasks, such as transferring data from the database to the application itself and pushing it out to the user's web browser, any speed improvements we make will not noticeably impact users. When we make performance improvements, it's critical we have metrics showing that we made an impact.

Happily, there's a wonderful benchmarking tool that can help us make decisions about what parts of our code to optimize and how to optimize them. Hugo Duncan's Criterium[1] is the benchmarking library of choice for many Clojure developers, and with good reason. Getting started with Criterium is

1. https://github.com/hugoduncan/criterium

• Click HERE to purchase this book now. discuss

pretty easy, since it's just a matter of adding it to your Leiningen dependencies. Criterium runs your code many times in order to take JIT and garbage collection into account. It also collects statistics on the running times to provide more detailed information, such as standard deviation and outliers. (Don't worry about the cryptic :jvm-opts ^:replace [] line—it just enables some JVM JIT optimizations.)

As an aside, the optimizations result in a slightly longer startup time, but it's worth taking that extra time in performance testing and other real-world production scenarios.

performance/project.clj

```
(defproject foo "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]]
  :jvm-opts ^:replace []
  :profiles {:dev {:dependencies [[criterium "0.4.2"]]}})
```

Criterium actually uses macros internally for some of the same reasons we discussed in Chapter 4, *Evaluate Code in Context*, on page ?, but here we just want to measure performance improvements. Let's pop open a REPL inside a project with project.clj and see how to use Criterium to measure the code's speed before optimizing:

performance/criterium_run.clj

```
user=> (use 'criterium.core)
user=> (defn length-1 [x] (.length x))
;=> #'user/length-1
user=> (bench (length-1 "hi there!"))
;Evaluation count : 26255400 in 60 samples of 437590 calls.
;             Execution time mean : 3.250388 µs
;    Execution time std-deviation : 850.690042 ns
;   Execution time lower quantile : 2.303419 µs ( 2.5%)
;   Execution time upper quantile : 5.038536 µs (97.5%)
;                   Overhead used : 2.193109 ns
;
;Found 1 outliers in 60 samples (1.6667 %)
;        low-severe       1 (1.6667 %)
; Variance from outliers : 94.6569 % Variance is severely inflated by outliers
;=> nil
```

The bench macro executes the given expression several times, so it may take a while before we see the output. We can wrap it with (with-progress-reporting ...) to see more feedback while the benchmarking system is running. From here on out, we'll only show the mean and standard deviation times, but you can try them out in your REPL if you want statistics on outliers and other details. If you have some logs to analyze, why not run that through Criterium and see how it goes?

A few microseconds might not sound like a lot, but when you're in the middle of a graph search algorithm or some other CPU-intensive task, it may be a significant bottleneck. We can pretty easily knock off some time by avoiding reflection (a Java mechanism that determines dynamically how a method will dispatch at runtime). Let's see what happens when we add a type hint (^String) to the argument to length-1, or even just call .length on the literal string directly, since these cases don't require reflection:

performance/no_reflection.clj

```
user=> (defn length-2 [^String x] (.length x))
;=> #'user/length-2
user=> (bench (length-2 "hi there!"))
;             Execution time mean : 1.476211 ns
;    Execution time std-deviation : 0.295418 ns

user=> (bench (.length "hi there!"))
;             Execution time mean : 3.423909 ns
;    Execution time std-deviation : 0.202517 ns
```

Figuring out where to put type hints to avoid reflection can be a little tricky, and since avoiding reflection is necessary for high-performance Clojure code, it's nice to have a hook to warn us about it: (set! *warn-on-reflection* true). If we set that var to true, we'll get a helpful warning letting us know that the code we just wrote will need to use reflection:

performance/warn_on_reflection.clj

```
user=> (set! *warn-on-reflection* true)
;=> true
user=> (defn length-1 [x] (.length x))
;Reflection warning, NO_SOURCE_PATH:1:20 -
;   reference to field length can't be resolved.
;#'user/length-1
```

There are all kinds of other tools to see how your Clojure code is performing, from profilers like JVisualVM[2] (included in the Oracle JDK distribution) to disassemblers like no.disassemble.[3] These tools are there to help you pick the right problems to solve, so be sure to use them! You don't want to waste your valuable time optimizing code that's already fast enough.

Hiding Performance Optimizations

Now that we've proven a particular bit of code is really too slow, what can we do to make it faster? Shantanu Kumar's *Clojure High Performance Programming [Kum13]* is full of techniques to improve the performance of Clojure programs

2. http://jvisualvm.java.net
3. https://github.com/gtrak/no.disassemble

and we touch upon two tricks in this chapter. Often these techniques mean embracing the wonderful Java interop of Clojure, using primitives, arrays, and type hinting to match Java performance. However, a codebase packed with features like these can make Clojure *feel* like Java, which can be shocking for a functional programming devotee. Let's look at how we can speed things up without winding up with ugly, unreadable code.

Imagine you've been working on a feature for one of your back-office applications that computes a few statistics on the latest sales numbers. You've tracked the biggest problem to a tight loop that adds a vector of numbers together, using a function called sum that's written in a pretty typical Clojure style. Here's how you might speed it up:

performance/sum.clj

```
(defn sum [xs]
  (reduce + xs))
(def collection (range 100))
(bench (sum collection))
;             Execution time mean : 2.078925 µs
;    Execution time std-deviation : 378.988150 ns

(defn array-sum [^ints xs]
  (loop [index 0
         acc 0]
    (if (< index (alength xs))
      (recur (unchecked-inc index) (+ acc (aget xs index)))
      acc)))
(def array (into-array Integer/TYPE (range 100)))
(bench (array-sum array))
;             Execution time mean : 161.939607 ns
;    Execution time std-deviation : 5.566530 ns
```

In array-sum, we've got a Java array, a type hint, and a low-level loop/recur instead of the much more elegant reduce version, but in exchange for that noise, we get about a 15x speedup. What do you think—is the speed worth all this code complexity? If you answered "It depends on the context," well done!

We won't try to cover every low-level speedup technique; instead you'll see how you can get the speed you want *and* the clarity you want. In this case, you can use a macro from clojure.core, areduce, to clean things up a bit:

performance/sum_with_areduce.clj

```
(defn array-sum [^ints xs]
  (areduce xs index ret 0 (+ ret (aget xs index))))

(bench (array-sum array))
;             Execution time mean : 170.214852 ns
;    Execution time std-deviation : 18.504698 ns
```

The clojure.core/areduce macro is implemented in the core language similarly to the previous version of array-sum, so it's no surprise that its performance is just as good.

performance/areduce_implementation.clj

```
(defmacro areduce
  "Reduces an expression across an array a, using an index named idx,
  and return value named ret, initialized to init, setting ret to the
  evaluation of expr at each step, returning ret."
  {:added "1.0"}
  [a idx ret init expr]
  `(let [a# ~a]
     (loop [~idx 0 ~ret ~init]
       (if (< ~idx  (alength a#))
         (recur (unchecked-inc ~idx) ~expr)
         ~ret))))
```

There's a library from Prismatic called hiphip[4] that goes a step further and eliminates the need for type hints.

After adding the Leiningen coordinates ([prismatic/hiphip "0.1.0"]) to your project.clj's :dependencies vector, you can try this out in your REPL as well (make sure to restart it after adding the dependencies):

performance/sum_with_hiphip.clj

```
(require 'hiphip.int)
(bench (hiphip.int/asum array))
;             Execution time mean : 144.535507 ns
;    Execution time std-deviation : 1.587751 ns
```

In hiphip's case, the big win in using a macro is that it can share nearly all of the implementation code across multiple data types (doubles, floats, ints, and longs). It even gets a little fancier, too. Because asum is a macro, it's also able to provide variants that allow some elegant bindings similar to the ones in for and doseq (see the hiphip.array docstring for details):

performance/sum_with_hiphip_fanciness.clj

```
(defn array-sum-of-squares [^ints xs]
  (areduce xs index ret 0 (+ ret (let [x (aget xs index)] (* x x)))))

(bench (array-sum-of-squares array))
;             Execution time mean : 1.419661 µs
;    Execution time std-deviation : 256.799353 ns

(bench (hiphip.int/asum [n array] (* n n)))
;            Execution time mean : 1.591465 µs
;    Execution time std-deviation : 232.503393 ns
```

4. https://github.com/prismatic/hiphip

The hiphip.int/asum version has pretty much identical performance to array-sum-of-squares. Meanwhile, it's a heck of a lot nicer to read and understand, right?

We don't have the space to go into all the details of hiphip's asum implementation, but here's the asum definition that's shared across the various data types:

performance/hiphip_asum.clj

```
;; from hiphip/type_impl.clj
(defmacro asum
  ([array]
     `(asum [a# ~array] a#))
  ([bindings form]
     `(areduce ~bindings sum# ~(impl/value-cast +type+ 0) (+ sum# ~form))))
```

The areduce here is hiphip's version that provides fancy bindings, not the areduce from clojure.core. impl/value-cast is what gives asum its ability to act on different data types based on the value of +type+ at compile time.

hiphip/type_impl.clj, where this macro is defined, is loaded directly from namespaces like hiphip.int via (load "type_impl"), instead of relying on the usual Clojure :require mechanism. Furthermore, hiphip/type_impl.clj doesn't actually define a namespace, so when it's loaded in hiphip.int, for example, it defines the asum macro in that namespace. This explains why +type+ can vary for the different types instead of just being a single unchanging value. This loading mechanism is a bit of a hack to avoid a macro-defining macro, which often means multiple levels of syntax-quoting. I can't fault anyone for wanting to avoid macro-defining macros—they're hard to write and even harder to understand later.

In Clojure's built-in areduce and hiphip's asum, we've seen macros enable great performance, comparable to loop/recur, while keeping the code concise and easy to understand. Next we'll look at a different approach to performance optimization that can allow us to sidestep problems entirely rather than improving them incrementally.