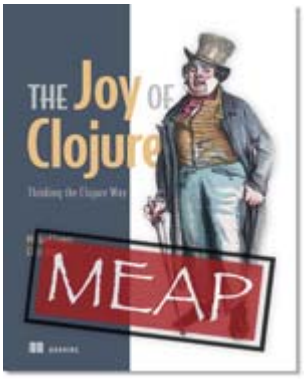
MANNING PUBLICATIONS

Designing a persistent toy

An article from



Joy of Clojure
EARLY ACCESS EDITION

Thinking the Clojure Way
Michael Fogus and Chris Houser
MEAP Release: January 2010
Softbound print: Fall 2010 | 300 pages
ISBN: 9781935182641

This article is taken from the book Joy of Clojure*. The authors discuss structural sharing using xconj and how it eliminates the need for full copy-on-write.*

We will not go into terrible detail about the internals of Clojure's persistent data structures—we'll leave that to others[1]. But we do want to explore the notion of structural sharing. Our example will be highly simplified compared to Clojure's implementations, but it should help clarify some of the techniques used.

The simplest shared-structure type is a list. Two different items can be added to the front of the same list, producing two new lists that share their `next` parts.

Let's try this out by actually creating a base list and then two new lists from that same base:

```
(def baselist (list :barnabas :adam))
(def lst1 (cons :willie baselist))
(def lst2 (cons :phoenix baselist))

lst1
;=> (:willie :barnabas :adam)

lst2
;=> (:phoenix :barnabas :adam)
```

We can think of `baselist` as a historical version of both `lst1` and `lst2`. But it's also the shared part of both lists. More than being equal, the `next` parts of both lists are *identical*—the very same instance:

[1] An excellent example being http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/

```
(= (next lst1) (next lst2))
;=> true

(identical? (next lst1) (next lst2))
;=> true
```

So that's not too complicated, right? But the features supported by lists are also pretty limited. Clojure's vectors and maps also provide structural sharing while allowing you to change values anywhere in the collection, not just on one end. The key is the tree structure each of these uses internally. To help demonstrate how a tree can allow interior changes and maintain shared structure at the same time, let's build one of our own.

Each node of our tree will have 3 fields: a value, a left branch, and a right branch. We'll just put them in a map, like this:

```
{:val 5, :l nil, :r nil}
```

That's the simplest possible tree—a single node holding the value 5, with empty left and right branches. This is exactly the kind of tree we want to return when a single item is added to an empty tree. To represent an empty tree, we'll just use `nil`. Let's write our own `conj` function to build up our tree, starting with just the code for this initial case:

```
(defn xconj [t v]
   (cond
     (nil? t) {:val v, :l nil, :r nil}))

(xconj nil 5)
;=> {:val 5, :l nil, :r nil}
```

Hey, it works! Not too impressive yet though, so we need to handle the case where an item is being added to a non-empty tree. Let's keep our tree in order by putting values less than a node's `:val` in the left branch, and other values in the right branch. That means we need a test like:

```
(< v (:val t))
```

When that's true, we need our new value `v` to go into the left branch, `(:l t)`. If this were a mutable tree, we'd *change* the value of `:l` to be our new node. Instead, let's build a *new* node, copying in the parts of the old node that don't need to change. Something like:

```
{:val (:val t),
 :l (insert-new-val-here),
 :r (:r t)}
```

This will be our new root node. Now, we just need to figure out what to put for `insert-new-val-here`. If the old value of `:l` is `nil`, we simply need a new single-node tree—we even have code for that already so we could use `(xconj nil v)`. But what if `:l` is not `nil`? In that case, we just want to insert `v` in its proper place within whatever tree `:l` is pointing to—so `(:l t)` instead of `nil`:

```
(defn xconj [t v]
  (cond
    (nil? t) {:val v, :l nil, :r nil}
    (< v (:val t)) {:val (:val t),
                    :l (xconj (:l t) v),
                    :r (:r t)}))

(def tree1 (xconj nil 5))
tree1
;=> {:val 5, :l nil, :r nil}

(def tree1 (xconj tree1 3))
tree1
```

```
;=> {:val 5, :l {:val 3, :l nil, :r nil}, :r nil}

(def tree1 (xconj tree1 2))
tree1
;=> {:val 5, :l {:val 3, :l {:val 2, :l nil, :r nil}, :r nil}, :r nil}
```

There—it's working. At least it seems to be! There's a lot of noise in that output making it difficult to read. Here's a little function to traverse the tree in sorted order, converting it to a seq that will print more succinctly:

```
(defn xseq [t]
  (when t
    (concat (xseq (:l t)) [(:val t)] (xseq (:r t)))))

(xseq tree1)
;=> (2 3 5)
```

Now we just need a final condition for handling the insertion of values that are *not* lower than the node value:

```
(defn xconj [t v]
  (cond
    (nil? t)       {:val v, :l nil, :r nil}
    (< v (:val t)) {:val (:val t),
                    :l (xconj (:l t) v),
                    :r (:r t)}
    :else          {:val (:val t),
                    :l (:l t),
                    :r (xconj (:r t) v)}))
```

Now that we've got the thing built, hopefully you understand well enough how it's put together that this demonstration of the shared structure will be unsurprising:

```
(def tree2 (xconj tree1 7))
(xseq tree2)
;=> (2 3 5 7)

(identical? (:l tree1) (:l tree2))
;=> true
```

No matter how big the left side of a tree's root node is, something can be inserted on the right side without copying, changing, or even examining the left side. All those values will be included in the new tree, along with the inserted value. This example demonstrates several features that it has in common with all of Clojure's persistent collections:

- Every change creates at least a new root node, plus new nodes as needed in the path through the tree to where the new value is being inserted.
- Values and unchanged branches are never copied, but references to them are copied from nodes in the old tree to nodes in the new one.
- This implementation is completely thread safe in a way that's easy to check—no object that existed before a call to `xconj` is changed in any way, and newly created nodes are in their final state before being returned. There is simply no way for any other thread or even any other functions in the same thread to see anything in an inconsistent state.

There are a few ways our example falls down, though, when compared to Clojure's rather more production-quality code. It:

- Is just a binary tree[2].
- Can only store numbers.

[2] Clojure hash-maps, hash-sets, and vectors all have up to 32 branches per node. This results in dramatically shallower trees, and therefore faster lookups and updates.

- Will overflow the stack if the tree gets too deep.
- Produces (via `xseq`) a non-lazy seq that will contain a whole copy of the tree.
- Can create unbalanced trees that will have worst case algorithmic complexity[3].

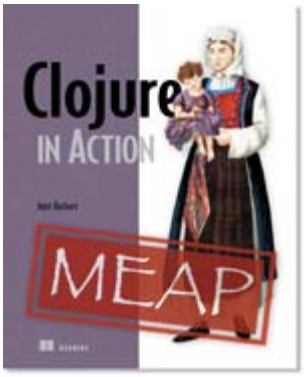
While structural sharing as described using `xconj` as a basis example can reduce the memory footprint of persistent data structures, it alone is insufficient. Instead, Clojure leans heavily on the notion of lazy sequences to further reduce its memory footprint.

Summary

Our implementation of a persistent sorted binary tree demonstrated how structural sharing eliminated the need for full copy-on-write. However, structural sharing is not enough to guarantee memory efficiency and that is where the benefits of laziness come into the fold.

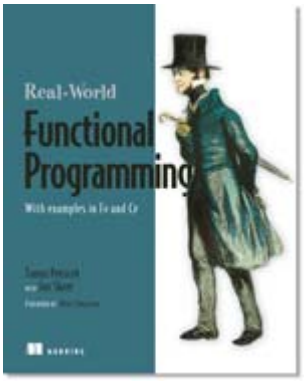
3 Clojure's sorted-map and sorted-set do use a binary tree internally, but implement red-black trees to keep the left and right sides nicely balanced.

Here are some other Manning titles you might be interested in:



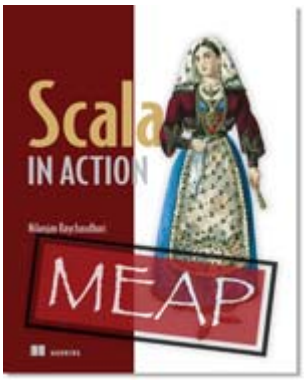
Clojure in Action
EARLY ACCESS EDITION

Amit Rathore
MEAP Release: November 2009
Softbound print: Winter 2010 | 475 pages
ISBN: 9781935182597



Real-World Functional Programming
IN PRINT

Tomas Petricek with Jon Skeet
December 2009 | 560 pages
ISBN: 9781933988924



Scala in Action
EARLY ACCESS EDITION

Nilanjan Raychaudhuri
MEAP Began: March 2010
Softbound print: Early 2011 | 525 pages
ISBN: 9781935182757