Programming in Declarative model

March 17, 2014
Overview

- Declarative programming in Clojure
  - typical functional programming (filter, map, fold)
  - recursion example
  - collection functions

- persistent vector and map

- trees
  - binary tree
    - depth-first traversal, breadth-first traversal
  - sorted binary tree
  - balanced sorted binary tree

- other algorithms: queues, graphs

- conclusions
  - expressiveness, readability, reasoning
Declarative programming in Clojure

Overview
Declarative programming in Clojure
Static scoping
Recursion example
Idiomatic functional operations
Sequence library in Clojure
Lazyness
Immutable data structures
Benefits
Persistent vector and map

Binary Tree

Binary search tree

Balanced BST

Other algorithms
Static scoping

Variables/parameters are only seen within scope

- **let** – define local symbols and execute statements
  
  ```clojure
  (let [bindings*] exprs*)
  ```

- **fn** – parameter symbols seen within function scope
  
  ```clojure
  (fn name? [params*] exprs*)
  ```

- **for** – local symbols seen with 'for' scope
  
  ```clojure
  (for [x (range 10)
        y (range 5)
        :let [z (+ x y)]
        :when (< z 10)]
    (list x y z))
  ```
Recursion example

```clojure
(def letterTypes #{Character/Lowercase_letter Character/Uppercase_letter})
(defn letter? [ch] (letterTypes (Character/getType ch)))

(defn scan-word [reader word]
  "Read single word from the reader."
  (let [ch (.read reader)]
    (if (letter? ch)
      (recur reader (cons ch word))
      (apply str (map char (reverse word))))))

(defn scan-words-impl [reader words]
  (let [ch (.read reader)] ; read next symbol
    (if (= ch -1) ; is end of reader?
      words ; finish and return words
      (if (letter? ch)
        ;; scan the word
        (recur reader (conj words (scan-word reader (list ch))))
        ;; skip character
        (recur reader words)))))

(defn scan-words [reader]
  (scan-words-impl reader []))
```

Idiomatic functional operations

- **filter** – filter each element of the list according to the predicate
  - list + predicate = list

- **map** – transform each element of the list
  - list + transform function = list

- **fold** (reduce) – reduce all elements of the list using given function
  - list + reduction function = scalar
  - \([X \ Y \ Z]\) and \(\ast\) as the reduction function (notation is not Clojure)
    - \((X \ast Y) \ast Z\) for left folding
    - \(X \ast (Y \ast Z)\) for right folding
Sequence library in Clojure

- conj, into
- range, repeat, repeatedly, iterate
- take, cycle
- interleave, interpose (join)
- constructors (list, vector, hash-set, hash-map)
  - compare to vec
- filter, take-while, drop-while, split-at, split-with
- every?, some
- map, reduce, sort, sort-by
- for – does map + filter
Lazyness

- sequence functions are lazy

Example
Immutable data structures

- “Purely functional data structures” (book)
  - immutable – whoever has reference never sees any changes
  - structural sharing – old instance shares data with the new one
  - asymptotically (almost) as efficient as transient implementations
    - $O(\log_{32} n)$ for unsorted collections (vector, hash-map, hash-set)
    - $O(\log_2 n)$ for sorted collections (sorted-map, sorted-set)
  - in Clojure referred as persistent collections

- transient counterparts
  - handy for local, temporary operations, like StringBuffer
Benefits

- may be safely
  - passed to functions
  - used as keys in maps and entries in sets

- enabling concurrency
  - sharing collections between threads is safe

- free versioning
  - old versions of the instance are untouched – may store them to a sequence and use for undo
Persistent vector and map

Overview
Declarative programming in Clojure
Persistent vector and map
Persistent vector
Persistent hashmap
Binary Tree
Binary search tree
Balanced BST
Other algorithms
Persistent vector

- Understanding Clojure’s PersistentVector implementation

- tree with 32 children, elements stored from leftmost to the right
  - 2-level may store $32^2 = 1024$ elements
  - 6-level may store $32^6 = 2^{30}$ elements
- expands as necessary – adding to the end is cheap
- tail data optimization – store tail to separate buffer until exceeds 32 elements
Overview
Declarative programming in Clojure
Persistent vector and map
Persistent vector
Persistent hashmap
Binary Tree
Binary search tree
Balanced BST
Other algorithms

Persistent hashmap

- Understanding Clojure’s PersistentHashMap
- same tree with 32 elements for hashed values
  - would be inefficient in terms of space (1 value adds at least 32 element node)
  - store only available values in smaller array
    - one 32-bit integer defines which values are present
    - index operations are needed to identify and locate stored values
      - count bits is most non-trivial one
Overview
Declarative programming in Clojure
Persistent vector and map

Binary Tree
Definition
Depth-first traversal
DFT with accumulator
DFT with state transition
DFT: sum with 2 threads
Breadth-first traversal
Concurrency in tree traversals

Binary search tree

Balanced BST

Other algorithms
A node in binary tree can be either

- a leaf – :leaf
- an internal node – with left and right subtrees

\[\text{defrecord Node [value left right]}\]

**Ex:** (Node. 15 :leaf (Node. 33 :leaf :leaf))

More compactly

```
  15
  / 
 33
```

```
  leaf
   
```

```
  leaf
   
```

```
  leaf
   
```

```
  leaf
   
```

```
  leaf
   
```

Depth-first traversal

- Traverse left subtree then right subtree

\[
\text{(defn dft [node]}
\begin{align*}
\text{(if (= node :leaf)} & \quad \text{(println :leaf)} \\
\text{\quad (do)} & \quad \text{(println (.value node))} \\
\text{\quad \quad (dft (.left node))} \\
\text{\quad \quad (dft (.right node))})
\end{align*}
\]

- by itself not specifically useful
  - add accumulator
DFT with accumulator

Overview
Declarative programming in Clojure

Persistent vector and map

Binary Tree
Definition
Depth-first traversal

DFT with accumulator
DFT with state transition
DFT: sum with 2 threads
Breadth-first traversal
Concurrency in tree traversals

Binary search tree

Balanced BST

Other algorithms

(defn dft-acc-impl [node acc]
  (if (= node :leaf)
    acc
    (let
      [lAcc (dft-acc-impl (.left node) acc)
       cAcc (cons (.value node) lAcc)
       rAcc (dft-acc-impl (.right node) cAcc)]
      rAcc)))

(defn dft-acc [node]
  (reverse (dft-acc-impl node '())))

- List accumulator
  - grow from list head
  - reverse at the end
DFT with state transition

- this is Oz: think in terms of \textit{reduce} for sequences
- just abstract accumulator as state transition

- $S_2$ and $S_3$ are “intermediate states”

```prolog
proc \{DFTState\ T \ P \ In \ ?Out\}
    proc \{DFTStateLoop\ T \ S1 \ ?Sn\}
        case T
        of leaf then Sn=S1
        [] tree(X T1 T2) then S2 S3 in
        S2=\{P \ S1 \ X\}
        \{DFTStateLoop\ T1 \ S2 \ S3\}
        \{DFTStateLoop\ T2 \ S3 \ Sn\}
    end
end
end
in
\{DFTStateLoop\ T \ In \ Out\}
end
```
DFT: sum with 2 threads

```clojure
fun {ExtendStream S X} S2 in
    S=X|S2
    S2 \% return new state
end

Start \% initial state: empty stream (unbound var)
Tree \% our tree (assume defined)
thread End in
    End=nil
    End={DFTState Tree ExtendStream Start}
end

Sum=thread {FoldL Start fun {\$ X Y} X+Y end 0}
end
```

- producer – consumer (one traverses, another sums)
- state is unbound tail list
  - grow from list tail
Breadth-first traversal

1. put tree root node to a queue
2. iteratively
   (a) take tree node from the queue
   (b) process node
   (c) put child nodes to the queue

- one possibility for queue implementation
  - difference list $QH \# QT$
  - use $\{\text{IsDet } QH\}$ to check if the queue is empty
    - if true, end iteration

\[
\text{proc } \{\text{BFTState } T \ P \ In \ ?Out\} \\
\text{proc } \{\text{BFTStateLoop } QH \# QT \ S \ ?Out\}
\]
Concurrency in tree traversals

- In DFT: definitional declarativeness
  - may use the results of traversal in other threads without risk (by reading list tail)
    - synchronization is automatic on list tail bind
  - may put `thread...end` statement anywhere

- In BFT: observational declarativeness
  - may use the results of traversal in other threads without risk
  - may occasionally have `race conditions` if put `thread...end` inside BFT procedure
    - happens because we have non-deterministic `{IsDet QH}`
Binary search tree

Overview
Declarative programming in Clojure
Persistent vector and map
Binary Tree
Binary search tree
Definition
Lookup
Insertion
Insertion (2)
Deletion (1)
Deletion (2)
Balanced BST
Other algorithms
Definition

- binary search tree (ordered binary tree)
  - values in
    - left subtree are smaller
    - right subtree are larger

![Tree Diagram]

- Definition
- Lookup
- Insertion
- Insertion (2)
- Deletion (1)
- Deletion (2)
- Balanced BST

Other algorithms
(defn lookup [node v]
  (if (= node :leaf)
    :not-found
    (let [nodeV (.value node)]
      (cond
       (= v nodeV) v
       (< v nodeV) (lookup (.left node) v)
       (> v nodeV) (lookup (.right node) v))))

- this is set, should have key&value to act as map
(defn insert-1 [node v]
  (if (= :leaf node)
    (Node. v :leaf :leaf)
    (let [{nv :value nl :left nr :right} node]
      (if (< v nv)
        (Node. nv (insert-1 nl v) nr)
        (Node. nv nl (insert-1 nr v))))))
(defn insert-1 [node v]
  (if (= :leaf node)
    (Node. v :leaf :leaf)
    (let [{nv :value nl :left nr :right} node]
      (if (< v nv)
        (Node. nv (insert-1 nl v) nr)
        (Node. nv nl (insert-1 nr v))))))
Record may have many fields – we want to redefine one
Record acts as a type (later) and as a dict

- use `assoc` as for dictionary
- record fields accessed by Keyword keys

```
(defn insert [node v]
  (if (= :leaf node)
    (Node. v :leaf :leaf)
    (if (< v (.value node))
      (assoc node :left (insert (.left node) v))
      (assoc node :right (insert (:right node) v))))
```
fun \{\text{RemoveSmallest} \ T\}\n  \text{case} \ T \\
  \text{of} \ \text{leaf} \ \text{then} \ \text{none} \\
  \text{}[] \ \text{tree}(Y \ T1 \ T2) \ \text{then} \\
  \text{case} \ \{\text{RemoveSmallest} \ T1\} \\
  \text{of} \ \text{none} \ \text{then} \ Y\#T2 \\
  \text{}[] \ Yp\#Tp \ \text{then} \ Yp\#\text{tree}(Y \ Tp \ T2) \\
  \text{end} \\
  \text{end} \\
  \text{end} \\

\begin{itemize}
  \item Y
  \item T1
  \item T2
  \item \text{?}
  \item Yp
  \item T1
  \item Tp
\end{itemize}
Deletion (2)

1. find \( Y \) in tree
2. cut \( Y_p \) (smallest) from right subtree
3. put \( Y_p \) and \( T_p \) in place

```clojure
fun {Delete X T}
  case T
    of leaf then leaf
    [] tree(T T1 T2) andthen X==Y then
      case {RemoveSmallest T2}
        of none then T1
        [] Yp#Tp then tree(Yp T1 Tp)
      end
    [] tree(T T1 T2) andthen X<Y then
      tree(Y {Delete X T1} T2)
    [] tree(T T1 T2) andthen X>Y then
      tree(T T1 {Delete X T2})
  end
end
```
Balanced BST
Balanced trees

- Balanced binary search tree
  - binary search tree
  - complexities
    - lookup $O(\log n)$
    - insertion $O(\log n)$
    - deletion $O(\log n)$
    - iterate over elements $O(n)$

- Variants
  - Red-black tree
  - AVL tree
  - others
Red-black tree: definition

1. A node is either red or black
2. Root node is black
3. All leaves are black
4. Both children of every red node are black
5. Same number of black nodes in each path from the root to a leaf

Property

- the longest path is less than 2 times the shortest path
  - hint: shortest must be all blacks
left and right rotation if two nodes are red

- assume changes are made in A (B)

right rotation

- if U and root of A (root of B) are both red
- change the color of the root A (root of B) to black
- preserves ordering
fun {RotateRight T}
    tree(C Z LT T4)=T
in
    case LT
    of tree(red Y tree(red X T1 T2) T3) then
        tree(red Y
            tree(black X T1 T2)
            tree(black Z T3 T4))
    [] tree(red X T1 tree(red Y T2 T3)) then
        tree(red Y
            tree(black X T1 T2)
            tree(black Z T3 T4))
    else T
    end
end
Rotate left: code

```
fun {RotateLeft T}
    tree(C X T1 RT)=T
in
    case RT
    of tree(red Z tree(red Y T2 T3) T4) then
        tree(red Y
            tree(black X T1 T2)
            tree(black Z T3 T4))
    [] tree(red Y T2 tree(red Z T3 T4)) then
        tree(red Y
            tree(black X T1 T2)
            tree(black Z T3 T4))
    else T
    end
end
```
Insertion (1)

- insert red node as in binary search tree
  - rotate if somewhere appear two red nodes
- leave black in the root (may appear red as the result of rotation)

```
fun {Insert X T}
  tree(C Y T1 T2)={InsertRot X T}
  in
  tree(black Y T1 T2)
end
```
fun {InsertRot X T}
  case T
  of leaf then tree(red X leaf leaf)
  [] tree(C Y T1 T2) andthen X==Y then T
  [] tree(C Y T1 T2) andthen X<Y then
    T3={InsertRot X T1} in
    {RotateRight tree(C Y T3 T2)}
  [] tree(C Y T1 T2) andthen X>Y then
    T3={InsertRot X T2} in
    {RotateLeft tree(C Y T1 T3)}
  end
end
Did not use any non-deterministic functions (like IsDet)
- we should have **definitional declarativeness**
  - can put `thread...end` anywhere
    - not of great help because we get new subtree only after finishing the old one
- **observational declarativeness** is still useful
  - we can traverse the tree and *concurrently* change it!
    - for example, update sorted GUI List from (sorted) RB tree without stopping to add new nodes to the tree
Other algorithms
Queue (1)

- queue may be represented as list, e.g. `1|4|5|nil`
  - adding or removing from head is easy and cheap
  - adding or removing from tail requires full list copy

- alternative: queue with 2 lists
  - Backward list is used to prepend elements to it
  - Forward list is used to take elements from it
  - `queue = FList + reversed BList`
  - if forward list is empty it is replaced with reversed backward list
fun {New} queue(nil nil) end

fun {IsEmpty Q}

  queue(FList BList) = Q

in

  FList==nil andthen BList==nil

end

% append element to the queue
proc {Append Q X ?Qn}

  queue(FList BList) = Q

in

  Qn = queue(FList X|BList)

end
% take element from the queue
proc {Take Q ?X ?Qn}
    queue(FList BList) = Q
in
    if FList\=nil then
        X = FList.1
        Qn = queue(FList.2 BList)
    elseif BList\=nil then RBLList in
        % reverse
        RBLList = {Reverse BList}
        X = RBLList.1
        Qn = queue(RBLList.2 nil)
    else
        raise app("List is empty") end
end
end
new reference is created for the updated queue

% append elements
Qu = {New}
Qu2 = {Append Qu 3}
Qu3 = {Append Qu2 1}
Qu4 = {Append Qu3 2}
{System.show Qu4}

% take element
X
Qu5 = {Take Qu4 X}
{System.show X}
{System.show Qu5}

easy *undo*

- store old references: immutable data structures make it possible (same for trees!)
Concurrent in queue

- The queue is safe (excluding exception throw)
  - Remind: Exceptions with concurrency create non-determinism
- No concurrent update is possible (same as with RB trees)
  - New queue “root” element on update
    - Concurrent update results in 2 different queues
- Concurrent traverse and update is still useful
In declarative model (as compared to non-declarative)

- traversing graph is possible
  - complexity (usually) multiple of $\log n$
  - unless graph node ids can be stored in array

- changing graph may require copying the whole graph
  - because of immutable data
  - node references
  - same problem for double linked list
Dynamic algorithms

- Some are possible with the same complexity
  - if our structures do not need to mutate
  - very restrictive