Clojure: Dynamic Functional Programming for the JVM™

Rich Hickey
http://clojure.org
Agenda

- Fundamentals
- Rationale
- Feature Tour
- Integration with the JVM
- Q&A
Clojure Fundamentals

> Dynamic
  • A new Lisp, not Common Lisp or Scheme
> Functional
  • Emphasis on immutability
> Supporting Concurrency
> Hosted on the JVM
  • Compiles to JVM bytecode
> Not Object-oriented
Why use a dynamic language?

> Flexibility
> Interactivity
> Concision
> Exploration
> Focus on your problem

> == Productivity
Why the JVM?

> VMs, not OSes, are the target platforms of future languages, providing:

• Type system
  • Dynamic enforcement and safety

• Libraries
  • Huge set of facilities

• Memory and other resource management
  • GC as platform, not language, facility

• Bytecode + JIT compilation
Why a Lisp?

> Dynamic
> Small core
> Elegant syntax
> Core advantage (still): code-as-data and syntactic abstraction
> Saw opportunities to reduce parens count
Why Functional?

> Easier to reason about
> Easier to test
> Essential for concurrency
> Few dynamic functional languages
  • Most focus on static type systems
> Functional by convention is not good enough
Why Focus on Concurrency?

> Multi-core is here to stay
> Multithreading a real challenge in Java et al
  • Locking is too hard to get right
> Functional programming and immutability helps
  • Share freely between threads
> But ‘changing’ state a reality for simulations and working models
> Automatic/enforced language support needed
Why not OO?

> OO encourages mutable state
  > Mutable stateful objects are the new spaghetti code
  > Hard to reason about
  > Disaster for concurrency
  > Encapsulation != concurrency semantics

> Common Lisp’s generic functions proved utility of methods outside of classes

> Polymorphism shouldn’t be based (only) on types
Feature Tour

- Lisp
- Data types and data abstractions
- Syntax
- Persistent Data Structures and Functional Programming
- Abstraction-based library
- Concurrent Programming
- JVM/Java Integration
Clojure is a Lisp

> Dynamically typed, dynamically compiled
> Interactive - read-eval-print-loop (REPL)
> Load/change code in running program
> Code as data
  • text -> reader -> data -> compiler
> Small core
> Sequences
> Syntactic abstraction - macros
Traditional evaluation
Clojure Evaluation

You

Code
Text

Reader

data structures
characters

Program

data structures

Program (macro)

data structures

Evaluator/Compiler

bytecode

JVM

Effect

Sun Microsystems
Atomic Data Types

> Arbitrary precision integers - 12345678987654
> Doubles 1.234 , BigDecimals 1.234M
> Ratios - 22/7
> Strings - "fred" , Characters - \a \b \c
> Symbols - fred ethel , Keywords - :fred :ethel
> Booleans - true false , Null - nil
> Regex patterns #“a*b”
Data Structures

> Lists - singly linked, grow at front
  • (1 2 3 4 5), (fred ethel lucy), (list 1 2 3)

> Vectors - indexed access, grow at end
  • [1 2 3 4 5], [fred ethel lucy]

> Maps - key/value associations
  • {:a 1, :b 2, :c 3}, {1 “ethel” 2 “fred”}

> Sets #{fred ethel lucy}

> Everything nests
> You’ve just seen it
> Data structures are the code
> Not text-based syntax
> • Syntax is in the interpretation of data structures
> Things that would be declarations, control structures, function calls, operators, are all just lists with op at front
> Everything is an expression
Syntax Comparison

> Control structures, function calls, operators, are all just lists with op at front:

<table>
<thead>
<tr>
<th>Java</th>
<th>Clojure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int i = 5;</code></td>
<td><code>(def i 5)</code></td>
</tr>
<tr>
<td><code>if(x == 0)</code></td>
<td><code>(if (zero? x)</code></td>
</tr>
<tr>
<td></td>
<td><code>y)</code></td>
</tr>
<tr>
<td></td>
<td><code>z)</code></td>
</tr>
<tr>
<td><code>x* y * z;</code></td>
<td><code>(* x y z)</code></td>
</tr>
<tr>
<td><code>foo(x, y, z);</code></td>
<td><code>(foo x y z)</code></td>
</tr>
<tr>
<td><code>file.close();</code></td>
<td><code>(.close file)</code></td>
</tr>
</tbody>
</table>
# Norvig’s Spelling Corrector in Python
# http://norvig.com/spell-correct.html

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))
alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    n = len(word)
    return set([word[0:i]+word[i+1:] for i in range(n)] +
                [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] +
                [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] +
                [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=lambda w: NWORDS[w])
(defn words [text] (re-seq #"[a-z]+" (.toLowerCase text)))

(defn train [features]
  (reduce (fn [model f] (assoc model f (inc (get model f 1))))
          {} features))

(def *nwords* (train (words (slurp "big.txt"))))

(defn edits1 [word]
  (let [alphabet "abcdefghijklmnopqrstuvwxyz", n (count word)]
    (distinct (concat
                (for [i (range n)] (str (subs word 0 i) (subs word (inc i))))
                (for [i (range (dec n))]
                    (str (subs word 0 i) (nth word (inc i)) (nth word i) (subs word (+ 2 i))))
                (for [i (range n) c alphabet] (str (subs word 0 i) c (subs word (inc i))))
                (for [i (range (inc n)) c alphabet] (str (subs word 0 i) c (subs word i))))))

(defn known [words nwords] (for [w words :when (nwords w)] w))

(defn known-edits2 [word nwords]
  (for [e1 (edits1 word) e2 (edits1 e1) :when (nwords e2)] e2))

(defn correct [word nwords]
  (let [candidates (or (known [word] nwords) (known (edits1 word) nwords)
                        (known-edits2 word nwords) [word]])
    (apply max-key #(get nwords % 1) candidates)))
Clojure is Functional

> All data structures immutable

> Core library functions have no side effects
  • Easier to reason about, test
  • Essential for concurrency

> let-bound locals are immutable

> loop/recur functional looping construct

> Higher-order functions
Persistent Data Structures

> Immutable, + old version of the collection is still available after 'changes'
> Collection maintains its performance guarantees for most operations
  • New versions are not full copies
  • Structural sharing key to efficiency
  • Thread safe, iteration safe
> All Clojure data structures persistent
  • Hash map/sets and vectors based upon array mapped hash tries (Bagwell)
Bit-partitioned hash tries
Abstraction-based Library

> Sequences, replace traditional Lisp lists
  • Seqs on all Clojure collections, all Java collections, Strings, regex matches, files...
  • Can be lazy - like generators

> All Collections

> Functions (call-ability)
  • Maps/vectors/sets are functions

> Many implementations
  • Extensible from Java and Clojure
Sequences

> Abstraction of traditional Lisp lists

> (seq coll)
  • If collection is non-empty, return seq object on it, else nil

> (first seq)
  • Returns the first element

> (rest seq)
  • Returns a sequence of the rest of the elements
Sequences

(drop 2 [1 2 3 4 5]) -> (3 4 5)

(take 9 (cycle [1 2 3 4])) -> (1 2 3 4 1 2 3 4 1)

(interleave [:a :b :c :d :e] [1 2 3 4 5]) -> (:a 1 :b 2 :c 3 :d 4 :e 5)

(partition 3 [1 2 3 4 5 6 7 8 9]) -> ((1 2 3) (4 5 6) (7 8 9))

(map vector [:a :b :c :d :e] [1 2 3 4 5]) -> ([:a 1] [:b 2] [:c 3] [:d 4] [:e 5])

(reduce + (range 100)) -> 4950
Maps and Sets

\[
\begin{align*}
&\text{(def m } \{a : 1 \ b : 2 \ c : 3\}) \\
&(m : b) \rightarrow 2 \ ;\text{also} \ (b \ m) \\
&(\text{keys m}) \rightarrow (a \ b \ c) \\
&(\text{assoc m} : d : 4 : c : 42) \rightarrow \{d : 4, \ a : 1, \ b : 2, \ c : 42\} \\
&(\text{merge-with} + m \{a : 2 \ b : 3\}) \rightarrow \{a : 3, \ b : 5, \ c : 3\} \\
&(\text{union} \\#\{a \ b \ c\} \#\{c \ d \ e\}) \rightarrow \#\{d \ a \ b \ c \ e\} \\
&(\text{join} \#\{a : 1 \ b : 2 \ c : 3\} \#\{a : 1 \ b : 21 \ c : 42\}) \\
&\quad \#\{a : 1 \ b : 2 \ e : 5\} \#\{a : 1 \ b : 21 \ d : 4\})
\end{align*}
\]

\[
\rightarrow \#\{d : 4, \ a : 1, \ b : 21, \ c : 42\} \#\{a : 1, \ b : 2, \ c : 3, \ e : 5\}
\]
Concurrency

> Interleaved/simultaneous execution
  • Must avoid seeing/yielding inconsistent data
> The more components there are to the data, the more difficult to keep consistent
> The more steps in a logical change, the more difficult to keep consistent
> Clojure also supports parallel computation
  • Emphasis here on coordination
Concurrency Methods

> Conventional way:
  • Direct references to mutable objects
  • Lock and worry (manual/convention)

> Clojure way:
  • Indirect references to immutable persistent data structures (inspired by SML’s ref)
  • Concurrency semantics for references
    • Automatic/enforced
    • No locks in user code

Lose the locks!
Typical OO - Direct references to Mutable Objects

- Unifies identity and value
- Anything can change at any time
- Consistency is a user problem
- Encapsulation doesn’t solve concurrency problems
Clojure - Indirect references to Immutable Objects

- Separates identity and value
- Obtaining value requires explicit dereference
- Values can never change
- Never an inconsistent value
- Encapsulation is orthogonal
Persistent ‘Edit’ 1 - Create new value

- New value is function of old
- Shares immutable structure
- Doesn’t impede readers
- Not impeded by readers
Persistent ‘Edit’ 2 - Atomic Update

- Always coordinated
- Multiple semantics
- Next dereference sees new value
- Consumers of values unaffected
Clojure References

- The only things that mutate are references themselves, in a controlled way

- 4 types of mutable references, with semantics for:
  - **Sharing** - change seen by multiple threads?
    - **Vars**: Not shared - isolated change within threads
  - **Synchrony** - change happens now/later?
  - **Coordination** - change multiple items at once?

<table>
<thead>
<tr>
<th></th>
<th>Independent</th>
<th>Coordinated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronous</strong></td>
<td>Atoms</td>
<td>Refs</td>
</tr>
<tr>
<td><strong>Asynchronous</strong></td>
<td>Agents</td>
<td>-</td>
</tr>
</tbody>
</table>
Refs and Transactions

> Software transactional memory system (STM)
> Refs can be changed only within a transaction
> All changes are Atomic and Isolated
  • Every change to Refs made within a transaction occurs or none do
  • No transaction sees the effects of any other transaction while it is running
> Transactions are speculative
  • Will be retried automatically if conflict
  • Must avoid side-effects!
The Clojure STM

- Surround code with (dosync ...)
- State changes through `alter/commute`, using ordinary function (state=>new-state)
- Uses Multiversion Concurrency Control (MVCC)
- All reads of Refs will see a consistent snapshot of the 'Ref world' as of the starting point of the transaction, + any changes it has made.
- All changes made to Refs during a transaction will appear to occur at a single point in the timeline.
(def foo (ref {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(assoc @foo :a "lucy")
-> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(alter foo assoc :a "lucy")
-> IllegalStateException: No transaction running

(dosync (alter foo assoc :a "lucy") ... other changes ...)  
@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
Java Integration

> Clojure strings are Java Strings, numbers are Numbers, collections implement Collection, fns implement Callable and Runnable etc
> Core abstractions, like seq, are Java interfaces
> Clojure seq library works on Java Iterables, Strings and arrays
> Implement and extend Java interfaces and classes
> Primitive arithmetic support equals Java’s speed
Java Interop

(.toUpperCase "fred")
-> "FRED"

(.getName String)
-> "java.lang.String"

(System/getProperty "java.vm.version")
-> "1.6.0_07-b06-57"

Math/PI -> 3.141592653589793

(doto (JFrame.) (.add (JLabel. "Hello World")) .pack .show)

; expands to:
(let [x (JFrame.)]
  (do (. x (add (JLabel. "Hello World")))
      (. x pack)
      (. x show))
  x)
(defn celsius []
  (let [frame (JFrame. "Celsius Converter")
        temp-text (JTextField.)
        celsius-label (JLabel. "Celsius")
        convert-button (JButton. "Convert")
        fahrenheit-label (JLabel. "Fahrenheit")]
    (.addActionListener convert-button
      (proxy [ActionListener] []
        (actionPerformed [evt]
          (let [c (Double/parseDouble (.getText temp-text))]
            (.setText fahrenheit-label
              (str (+ 32 (* 1.8 c)) " Fahrenheit"))))))
  (.doto frame
    (.setLayout (GridLayout. 2 2 3 3))
    (.add temp-text) (.add celsius-label)
    (.add convert-button) (.add fahrenheit-label)
    (.setSize 300 80) (.setVisible true))))
Benefits of the JVM

> Focus on the language vs code generation or reinventing libraries
> Sharing GC and type system with implementation/FFI language is huge benefit
> Tools - e.g. breakpoint/step debugging, profilers...
> Libraries! Users can do UI, database, web, XML, graphics, etc right away
> Great MT infrastructure
  • java.util.concurrent

Keep your Java™ investment!
There’s much more...

- Metadata
- Recursive functional looping
- Destructuring binding in `let/fn/loop`
- List comprehensions (`for`)
- Relational set algebra
- Multimethods
- Parallel computation
- Namespaces, zippers, XML ...
Why Clojure?

> Expressive, elegant, simple
  • Approachable functional programming
  • Robust, easy-to-use concurrency
> Powerful extensibility, good performance
> Leverage an established, accepted platform
> Good tools
  • NetBeans, IntelliJ, Emacs, YourKit ...
> Good documentation, great community
Rich Hickey
http://clojure.org/
http://groups.google.com/group/clojure