Clojure

A Dynamic Programming Language for the JVM

Rich Hickey
Agenda

- Fundamentals
- Rationale
- Feature Tour
- Experiences on 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 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 is platform, not language, facility
  - Bytecode + JIT compilation
Language as platform vs. Language + platform

- Old way - each language defines its own runtime
  - GC, bytecode, type system, libraries etc
- New way (JVM, .Net)
  - Common runtime independent of language
- Platforms are dictated by clients
  - Huge investments in performance, scalability, security, libraries etc.
Java/JVM is language + platform

- Not the original story, but other languages for JVM always existed, now embraced by Sun
- JVM has established track record and trust level
  - Now open source
- Interop with other code always required
  - C linkage insufficient these days
  - Ability to call/consume Java is critical
- Clojure is the language, JVM the platform
Why a Lisp?

- Dynamic
- Small core
  - Clojure is a solo effort
- Elegant syntax
- Core advantage still code-as-data and syntactic abstraction
- Saw opportunities to reduce parens-overload
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
• FP/Immutability helps
  • Share freely between threads
• But ‘changing’ state a reality for simulations and working models
• Automatic/enforced language support needed
Why not OO?

• Encourages mutable State
  • Mutable stateful objects are the new spaghetti code
  • Encapsulation != concurrency semantics
  • Common Lisp’s generic functions proved utility of methods outside of classes
  • Polymorphism shouldn’t be based (only) on types
  • Many more...
Feature Tour

- Data types and data abstractions
- Syntax
- Persistent Data Structures
  - Functional Programming
- Abstraction-based library
- Concurrent Programming
  - Transactions and Agents
- JVM/Java Integration
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
Syntax

• 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
<table>
<thead>
<tr>
<th>Java</th>
<th>Clojure</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 5;</td>
<td>(def i 5)</td>
</tr>
<tr>
<td>if(x == 0)</td>
<td>(if (zero? x)</td>
</tr>
<tr>
<td>return y;</td>
<td>y</td>
</tr>
<tr>
<td>else</td>
<td>z)</td>
</tr>
<tr>
<td>return z;</td>
<td></td>
</tr>
<tr>
<td>x* y * z;</td>
<td>(* x y z)</td>
</tr>
<tr>
<td>foo(x, y, z);</td>
<td>(foo x y z)</td>
</tr>
<tr>
<td>file.close();</td>
<td>(.close file)</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])
; Norvig’s Spelling Corrector in Clojure
; http://en.wikibooks.org/wiki/Clojure_Programming#Examples

(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)))
Persistent Data Structures

• Immutable, + old version of the collection is still available after 'changes'

• Collection maintains its performance guarantees
  • Therefore new versions are not full copies

• Structural sharing - thread safe, iteration safe

• All Clojure data structures are persistent
  • Hash map/set and vector based upon array mapped hash tries (Bagwell)
  • Practical - much faster than $O(\log N)$
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 seq of the rest of the elements, or nil if no more
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])

(apply str (interpose \, "asdf"))
-> "a,s,d,f"

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

(def m {,:a 1 ,:b 2 ,:c 3})

(m ,:b) -> 2 ;also (:b m)

(keys m) -> (:a ,:b ,:c)

(assoc m ,:d 4 ,:c 42) -> {,:d 4 ,,:a 1 ,,:b 2 ,,:c 42}

(merge-with + m {,:a 2 ,:b 3}) -> {,:a 3 ,,:b 5 ,,:c 3}

(union #{,:a ,:b ,:c} #{,:c ,:d ,:e}) -> #{,:d ,:a ,:b ,:c ,:e}

(join #{#{,:a 1 ,:b 2 ,:c 3} #{,:a 1 ,:b 21 ,:c 42}}
     #{#{,:a 1 ,:b 2 ,:e 5} #{,:a 1 ,:b 21 ,:d 4}})

-> #{#{,: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
Concurrent 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!
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’

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

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

```
foo
```

```
| :a  | "fred" |
| :b  | "ethel" |
| :c  | 42      |
| :d  | 17      |
| :e  | 6       |
```

```
| :a  | "lucy"  |
| :b  | "ethel" |
| :c  | 42      |
| :d  | 17      |
| :e  | 6       |
```
Clojure References

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

- 3 types of mutable references, with different semantics:
  -Refs - Share synchronous coordinated changes between threads
  -Agents - Share asynchronous autonomous changes between threads
  -Vars - Isolate changes within threads
Refs and Transactions

• Software transactional memory system (STM)
• Refs can only be changed 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 ...)`

• 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.

• Readers never impede writers/readers, writers never impede readers, supports commute
Refs in action

(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}

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

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

• Manage independent state

• State changes through actions, which are ordinary functions (state=>new-state)

• Actions are dispatched using send or send-off, which return immediately

• Actions occur asynchronously on thread-pool threads

• Only one action per agent happens at a time
 Agents

- Agent state always accessible, via `deref/@admin`, but may not reflect all actions
- Can coordinate with actions using `await`
- Any dispatches made during an action are held until `after` the state of the agent has changed
- Agents coordinate with transactions - any dispatches made during a transaction are held until it commits
- Agents are not Actors (Erlang/Scala)
Agents in Action

(def foo (agent {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

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

(send foo assoc :a "lucy")

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

(await foo)

/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

• New primitive arithmetic support equals Java’s speed.
Java Interop

Math/PI
3.141592653589793

(.. System getProperties (get "java.version"))
"1.5.0_13"

(new java.util.Date)
Thu Jun 05 12:37:32 EDT 2008

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

; expands to:
(let [x (JFrame.)]
  (do (. x (add (JLabel. "Hello World")))
      (. x pack)
      (. x pack)
      (. x show))
  x)
import '(javax.swing JFrame JLabel JTextField JButton)'
'(java.awt.event ActionListener) '(java.awt GridLayout))

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)))
Experiences on the JVM

• Main complaint is no tail call optimization
• HotSpot covers the last mile of compilation
  • Runtime optimizing compilation
• Clojure can get ~1 gFlop without even generating JVM arithmetic primitives
• Ephemeral garbage is extremely cheap
• Great performance, many facilities
  • Verifier, security, dynamic code loading
Benefits of the JVM

• Focus on my language vs code generation or mundane libraries

• Sharing GC and type system with implementation/FFI language is huge benefit

• Tools - e.g. breakpoint/step debugging etc.

• Libraries! Users can do UI, database, web, XML, graphics, etc right away

• Great MT infrastructure - java.util.concurrent

• well-defined memory model
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
- Approachable functional programming
- Robust, easy-to-use concurrency
- Powerful extensibility
- Good performance
- Leverage an established, accepted platform
- Good documentation
- Growing community
Thanks for listening!

http://clojure.org

Questions?