Clojure

A Dynamic Programming Language for the JVM

Rich Hickey

Agenda

- Introduction
- Fundamentals
- Rationale
- Feature Tour
- Status
- Q&A

Introduction

- Who am I?
- Who are you?
  - Know/use Lisp?
  - Java?
  - ML/Haskell?

Clojure Fundamentals

- Functional
  - Mostly
- Lisp
  - *Not* CL or Scheme
- Hosted
  - On the JVM
- Supporting Concurrency
- Open Source

Rationale

- Why yet another Language/Lisp?
- No existing language has desired fundamentals
  - Functional
  - Lisp
  - JVM Hosted
  - Thread-aware

Lisp is Good

- Often emulated/pillaged, still not duplicated
- Lambda calculus yields an extremely small core
- Almost no syntax
- Core advantage still code-as-data and syntactic abstraction

What about the standard Lisps (Common Lisp and Scheme)?

- Slow/no innovation post standardization
- Core data structures mutable, not extensible
- No concurrency in specs
- Good implementations already exist for JVM (ABCL, Kawa, SISC et al)
- Standard Lisps are their own platforms

Clojure is a Lisp not constrained by backwards compatibility

- Extends the code-as-data paradigm to maps and vectors
- Defaults to immutability
- Core data structures are extensible abstractions
- Embraces a platform (JVM)

Functional Programming is Good

- Immutable data + first-class functions
- Could always be done in Lisp, by discipline/ convention
  - But if a data structure can be mutated, dangerous to presume it won't be
  - In traditional Lisp, only the list data structure is structurally recursive
- Pure functional languages tend to strongly static types
  - Not for everyone, or every task

Clojure is a functional language with a dynamic emphasis

- All data structures immutable & persistent, supporting recursion
- Dynamic typing
  - Heterogeneous collections, argument and return types
- Runtime polymorphism

Languages and Platforms

- 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 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 also 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

Object Orientation is Overrated

- Born of simulation, now used for everything, even when inappropriate
  - Encouraged by Java/C# in all situations, due to their lack of (idiomatic) support for anything else
- Mutable stateful objects are the new spaghetti code
  - Hard to understand, test, reason about
  - Concurrency disaster
- Inheritance is *not* the only way to do polymorphism

Polymorphism is Good

- Switch statements, structural matching etc yield brittle systems
- Polymorphism yields extensible, flexible systems
- Clojure multimethods decouple polymorphism from OO and types
  - Supports multiple taxonomies
  - Dispatches via static, dynamic or external properties, metadata, etc

Concurrency and state in the multi-core future

- Immutability makes much of the problem go away
  - Share freely between threads
- But changing state a reality for simulations and for in-program proxies to the outside world
- Locking is too hard to get right over and over again
- Clojure's software transactional memory does the hard part

Feature Tour

- Dynamic Development
- Lisp
- Functional Programming
- Runtime Polymorphism/Multimethods
- Concurrent Programming/STM
- JVM/Java Integration

Dynamic Development

- REPL
- Basics
  - Clojure has arbitrary precision integers, strings, ratios, doubles, characters, symbols, keywords.
- Dynamic Compilation
  - To JVM bytecode

Lisp

- Reader
  - Code-as-data
  - Extended for maps and vectors
- Macros

```
(defmacro and
  ([] :t)
  ([x] x)
  ([x & rest] `(if ~x (and ~@rest))))
```

Functional Programming

- First-class functions

```
(defn make-adder [x]
      (let [y x]
        (fn [z] (+ y z))))
(def add2 (make-adder 2))
(add2 4)
-> 6
```

- Variable arity

```
(defn argcount
    ([] 0)
    ([x] 1)
    ([x y] 2)
    ([x y & more]
      (+ (thisfn x y) (count more))))

(argcount 1 2 3 4 5)
-> 5
```

Immutable Data Structures

- Lists, vectors, maps
- Reader, backquote support

```
(let [vec [1 2 3 4]
      map {:fred "ethel"}
      lst (list 4 3 2 1)]
    (list
        (conj vec 5)
        (assoc map :ricky "lucy")
        (conj lst 5)
        ;the originals are intact
        vec
        map
        lst))
-> ([1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (5 4 3 2 1)
    [1 2 3 4] {:fred "ethel"} (4 3 2 1))
```

Persistent Data Structures

- Not DB persistence
- Old version of the collection is still available after the 'change'
- Collection maintains its performance guarantees for most operations
  - Therefore new versions are not full copies
- Hash map and vector both based upon array mapped hash tries (Bagwell)
- Sorted map is red-black tree

Extensible Abstractions

- Clojure uses Java interfaces to define its core data structures
- Supports extension to new concrete implementations
  - Library functions will work with these extensions
- Big improvement vs. hardwiring a language to the concrete implementations of its data types

Example: The Seq Abstraction

- An abstraction of *cons*
  - *first* and *rest* functions
  - Immutable and persistent
  - Can be implemented lazily
- Many implementations
  - All collections provide seqs
  - Can get seqs over Java Iterables and arrays
- Library functions (map reduce filter etc) work with all seqs

Lazy Seqs

- *rest* not produced until requested
- Define your own lazy seq-producing functions using the *lazy-cons* macro
- Seqs can be used like iterators or generators of other languages

```
;the library function take
(defn take [n coll]
  (when (and (pos? n) (seq coll))
    (lazy-cons (first coll) (take (dec n) (rest coll)))))

;cycle produces an 'infinite' seq!
(take 15 (cycle [1 2 3 4]))
-> (1 2 3 4 1 2 3 4 1 2 3 4 1 2 3)
```

Metadata

- Orthogonal to the logical value of the data
- Symbols and collections support a metadata map
- Does not impact equality semantics
- Not seen in operations on the value
- Support for literal metadata in reader

```
(def vec [1 2 3])
(def attributed-vec (with-meta vec {:source :trusted}))
(:source ^attributed-vec)
-> :trusted
(eql? vec attributed-vec)
-> :t
```

Recursive Loops

- No mutable locals in Clojure
- No tail recursion optimization in the JVM
- *recur* op does constant-space recursive looping
- Rebinds and jumps to nearest *loop* or function frame

```
(defn zipmap [keys vals]
  (loop [map {}, ks (seq keys), vs (seq vals)]
    (if (and ks vs)
        (recur (assoc map (first ks) (first vs))
               (rest ks)
               (rest vs))
       map)))

(zipmap [:a :b :c] [1 2 3])
-> {:b 2, :c 3, :a 1}
```

Polymorphism via Multimethods

- Full generalization of indirect dispatch
  - Not tied to OO or types
- Fixed dispatch function which is an arbitrary function of the arguments
- Open set of methods associated with different values of the dispatch function
- Call sequence:
  - Call dispatch function on args to get dispatch value
  - Find method associated with dispatch value
    - else call default method if present
      - else error

Example: Multimethods

```
(defmulti encounter (fn [x y] [(:Species x) (:Species y)]))

(defmethod encounter [:Bunny :Lion] [b l] :run-away)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)

(def b1 {:Species :Bunny :other :stuff})
(def b2 {:Species :Bunny :other :stuff})
(def l1 {:Species :Lion :other :stuff})
(def l2 {:Species :Lion :other :stuff})

(encounter b1 b2)
-> :mate
(encounter b1 l1)
-> :run-away
(encounter l1 b1)
-> :eat
(encounter l1 l2)
-> :fight
```

Concurrent Programming

- Immutable persistent data structures
- But, in most practical programs some state changes
- Refs
  - Sharing changes between threads
  - Transactional
- Vars
  - Isolating 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!

Example: Refs

```
(import '(java.util.concurrent Executors))

(defn test-stm [nitems nthreads niters]
  (let [refs  (map ref (replicate nitems 0))
        pool  (. Executors (newFixedThreadPool nthreads))
        tasks (map (fn [t]
                     (fn []
                       (dotimes n niters
                         (sync nil
                           (dolist r refs
                             (set r (+ @r t)))))))
                   (range nthreads))]
      (. pool (invokeAll tasks))
      (. pool (shutdown))
      (map deref! refs)))

(test-stm 10 10 10000)
-> (550000 550000 550000 550000 550000 550000 550000 550000
550000 550000)
```

Vars

- Like CL's special vars
  - dynamic scope
  - stack discipline
- Shared root binding established by *def*
  - root can be unbound
- Can be *set!* but only if first thread-locally bound using *binding* (not *let*)
- Functions stored in vars, so they too can be dynamically rebound
  - context/aspect-like idoims

JVM/Java Integration

- Consuming Java
  - Instantiate objects with *new*
  - Access members with dot operator
    - Inference + hints can avoid reflection
  - Import class names with *import*
- Extending Java
  - Implement interfaces dynamically with *implement* macro

Example: Swing App

```
(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener)
        '(java.awt GridLayout))
(defn celsius []
  (let [frame (new JFrame "Celsius Converter")
        temp-text (new JTextField)
        celsius-label (new JLabel "Celsius")
        convert-button (new JButton "Convert")
        fahrenheit-label (new JLabel "Fahrenheit")]
    (. convert-button
        (addActionListener
          (implement [ActionListener]
             (actionPerformed [evt]
               (let [c (. Double (parseDouble (. temp-text (getText))))]
                 (. fahrenheit-label
                    (setText (strcat (+ 32 (* 1.8 c)) " Fahrenheit"))))))))
    (doto frame
               (setLayout (new GridLayout 2 2 3 3))
               (add temp-text) (add celsius-label)
               (add convert-button) (add fahrenheit-label)
               (setSize 300 80) (setVisible :t))))
(celsius)
```

JVM/Java Integration

- Clojure strings are Java Strings
- Clojure collections implement (read-only portion of) Collection interface
- Clojure fns are Callable
- Symbols and Keywords are Comparable
- Emits standard JVM debug info

Status

- Currently alpha
  - Download available
  - Everything I've talked about works
  - Some names might change
- Decent docs on web site
  - More coming
- Debugging with JSwat works pretty well
  - Breakpoints/Single-stepping
  - Locals

To Do

- More library functions
- Support for extending classes
- Editor/IDE support
  - Syntax highlighting, paren matching etc
  - Debugging
    - Java IDEs don't fully support debugging non-Java
- Fit/finish, performance tuning

More info

- Main site:
  - http://clojure.sourceforge.net/
- Discussion group:
  - http://groups.google.com/group/clojure

Thanks!

Questions?