Clojure intro

Adrian Muresan, Ovidiu Deac

Outline

- What? Why? How?
- Date types
- Functions & higher-order functions
- Mutable state & STM
- Java interop
- Performance
- Macros
- Tools & frameworks

What?

- A Lisp
  - LISt Processing
- On top of the JVM (and .NET runtime)
  - Interop with JVM is trivial
- Immutable state
  - Great for concurrency
  - Can affect performance
- Dynamic language
  - Good for productivity
  - Bad for performance (more details later)

Why?

- Why a Lisp?
  - Trivial syntax
    - Code is a list of terms
  - Expressive
    - Functional programming language
    - High-order functions
    - Lisp is its own meta-language (more details later)
    - Dynamic language
- Why the JVM?
  - Portable
  - Lots of libraries & frameworks

How?

- Compile to java .class
  - Object types are inferred at compile time
  - RTTI when types cannot be inferred
- Can eval code at runtime
  - REPL
  - Usually a bad idea to use eval

Data types

- Numeric – ints, floats, decimal, rational
- String, char, regex
- bool
- Keywords
  - Constants, values
  - Useful for indexing
  - Ex: :the-key

Data types

- Symbols
  - Ex: +, java.lang.String
  - (def my-const 42) → my-const is a symbol
  - (defn my-fn []) → my-fn is a symbol

Data types

- Lists
  - Collection of stuff
  - Ex: (1 2 3), (+ 4 5 6), (:k 42 “test”)
- Vectors
  - Ex: [“a” “b” “c”], [\c “s” 42]
- Sets
  - unique values inside
  - Ex: #{“a” “b” “c”}

Data types

- Maps
  - key-value pairs
  - Ex: (def info {:name “adi” :height 178})
  - (:name info) → “adi”
  - (get info :height) → 178
  - (assoc info :shoesize 43) → {:name “adi” :height 178 :shoesize 43}
- Any Java data type

Data types

- Records
  - Aggregate data types
- Reference types (more details later)
  - Atoms
  - Vars
  - Refs
  - Agents

Form evaluation

- Forms! Forms everywhere!
  - Everything is a form
  - They evaluate to something
- Examples
  - (list 1 :a “b”) → (1 :a “b”)
  - (+ 2 3) → 5
  - (println (+ 2 3)) → (println 5) → nil
  - (apply + (list 1 2 3)) → (+ 1 2 3) → 6
  - (apply str (list 1 :a “b”)) → “1:ab”

Form evaluation

- Can delay evaluation to on-demand – laziness
  - lazy-seq, map, concat
  - delay, force
- Examples
  - (take 6 (range 10000000000)) → (0 1 2 3 4 5)
- Attention
  - Laziness evaluates in chunks of 32
  - Have no side-effects inside
  - Don't assume anything about laziness evaluation

Functions

- Simple functions
  - (defn adder “my adder” [x y] (+ x y))
- Anonymous functions
  - (fn [x y] (+ x y))
  - #(+ %1 %2)
- Variable args
  - (defn mult-add [a1 & rest]

        (* a1 (apply + rest)))
  - (mult-add 10 20 30) → 500
- Ex: “+” is a function: (+ 1 2 3 4 5), (< 1 2 3 4 5)

Functions

- Doc-string

  -
    ```
    (defn func “description of func” [x y]
        (* x y))
    ```
- Pre/post conditions

  -
    ```
    (defn sqr [x]
        {:pre  [(pos? x) (integer? x)]
        :post [(= x (* % %))]}
        (* x x))
    ```

Destructuring arguments

- In functions
  - (defn f [{name :name height :height}]

    (str name “-” height))
  - (f {:name “adi” :height height}) → adi-178
- In let forms
  - (let [[v1 v2] [42 24]]

    (max v1 v2)) → 42
- Possible anywhere there are bindings
  - Fn args, let, for, doseq, loop, etc

Higher-order functions

- Compose – function composition
  - Useful when aggregating computation
  - (f (g x))
  - (map (comp f g) s) → seq of (f (g x)) | x in s
- Partial – partial function application
  - Useful when successively specializing a function
  - Ex: we have (defn query [name date] …)
    - (def get-mine (partial query “adi”)) → a fn

Higher-order functions

- Apply
  - Useful when function arguments come in a seq
  - Ex: we have a vector of numbers to sum
    - (apply + [1 2 3 4 5]) → 15
    - (apply bin-fn (list 1 2 3)) → exception (

Higher-order functions

- Filter
  - Useful for filtering sequences
  - Ex:
    - (def ppl

        (list {:name “Adi” :height 176} {:name “Ovidiu” :height 180}))
  - filter out all people under 180
  - (filter

    (fn [{height :height}] (>= height 180))

    ppl) → ({:name “Ovidiu” :height 180})

Higher-order functions

- Map
  - Useful for computations w/ seqs as inputs and outputs
  - Ex: Let's change the names of ppl by a “random” format function

  -
    ```
    (map
        (fn [p transf] (assoc p :name (transf (:name p))))
        ppl
        [to-lower to-upper])
    → ({:name “adi” :height 176} {:name “OVIDIU” :height 180})
    ```

Higher-order functions

- Reduce
  - Computing a single value from a sequence
  - Get the average age of our group

```
(defn red-fn [[sum cnt] pers]
    [(+ sum (:height pers)) (inc cnt)])
(let [[sum total] (reduce redfn [0 0] ppl)]
    (/ sum total))
```

References

- Atoms - CAS operations
- Agents - queued operations
- Ref - multiversion concurrency control

Atoms

- Synchonous objects
  - atomic operations on a reference
  - atom, swap!, reset!
  - Asynchonous actions
  - Queue of operations on a reference
  - Run on separate threads

Agents

- Asynchonous actions
- Queue of operations on a reference
- Run on separate threads

STM

- Similar to DB transactions
- Implementation of MVCC for memory locations
  - tuning with :min-history :max-history
- ACI (atomic, consistent, isolated)
- lock-free algorithms

STM 2

- Inside the transaction - pure functions ONLY
  - io! - guards for side effects
  - agent dispatch only at transaction commit

STM 3

- Basic operations
  - define reference with ref
  - build transaction with dosync

STM problems

- Write skew - use function ensure
- live lock - STM implements barging
- large transactions
- impure functions in a transaction

Java interop

- Instantiate a class
  - (def now (Date.))
  - (def my-map (new java.util.HashMap))
- Calling methods
  - (.toString now)
  - (. now toString)
- Calling static method
  - (. java.lang.System/out println "stuff")

Java interop

- Implement an interface / extend a baseclass
  - :gen-class – gen one class from current module
  - Defrecord – a new named datatype
  - Proxy – in-place, anonymous

Performance considerations

- Clojure (usually) slower than Java - can be improved
  - Always measure – remember the 80-20 rule
  - Insert type hints – avoid RTTI calls
  - Use transients (mutables)
  - Code the slow parts in Java & use interop

Macros

- What is a LISP program internally? A list.
- So what?
  - So we can modify it before compiling it
  - So we can introduce our own language abstractions
- What are Lisp macros?
  - “functions” that manipulate the program at compile time (important!)
  - A meta-language in Lisp
  - A way of building “language templates”
- What's the equivalent in Java / C#?

Useful macros

- For

  -
    ```
    (for [x [1 2]
              y [“a” “b”]]
         [x y]) → ([1 “a”] [1 “b”] [2 “a”] [2 “b])
    ```
- Doseq

  -
    ```
    (doseq
         [fruit (list “apples” “oranges”)]
         (println fruit))
    ```

Useful macros

- ->, ->>
  - (str (trim “ fruit “) “-ness”)
  - (-> “fruit”

        (trim)

        (str “-ness”)
- Doto
  - (doto (new java.util.HashMap)

        (.put "a" 1)

        (.put "b" 2))

Tools

- IDEs
  - LightTable, Eclipse, vim, emacs, etc
- Profiling
  - JVM Monitor, Jconsole, etc.
- Managing project dependencies
  - Leiningen
- All tools built for Java

Frameworks

- Web
  - Noir, Ring, Netty
- Gamedev
  - Play-clj (libgdx)
- Android
  - Lein-droid
- Music
  - Overtone

- Logic programming
- Datalog queries

- And probably a lot more that I don't know about

Demo

Questions?