Speaking Data:
Simple, Functional Programming with Clojure

Paul deGrandis :: @ohpauleez
Overview

● Clojure in **ten ideas**
  ○ “The key to understanding Clojure is ideas, not language constructs” - Stu Halloway
  ○ Everything I say about Clojure is true for ClojureScript too

● Software engineering with Clojure
  ○ Why Clojure? Why now?
  ○ Community / Ecosystem / Support

● Clojure **applied**
  ○ Walmart eReceipts / “Savings Catcher”
  ○ Boeing 737 MAX diagnostics system
  ○ DRW Trading
Overview: Clojure

- Getting Started / Docs / Tutorials - [https://clojure.org/](https://clojure.org/)
- A Lisp dialect (Lisp-1), small core, code-as-data
- Functional, emphasis on immutability, a language for data manipulation
- Symbiotic with an established platform
- Designed for concurrency, managed state
- Compiled, strongly typed, dynamic
- Powerful polymorphic capabilities
- Specifications are first-class
- Clojure programs are composed of expressions
Extensible Data Notation (edn)

- Extensible data format for the conveyance of values
- Rich set of built-in elements, generic dispatch/extension character
  - Domain can be fully captured and expressed in extensions
- Extensions to the notation are opt-in
- Clojure programs are expressed in edn; Serializable form of Clojure
- [https://github.com/edn-format/edn](https://github.com/edn-format/edn)
Extensible Data Notation (edn)

```json
{
  :firstName "John",
  :lastName "Smith",
  :age 25,
  :address {
    :streetAddress "21 2nd Street",
    :city "New York",
    :state "NY",
    :postalCode "10021"
  },
  :phoneNumber
  [ {
    :type "name",
    :number "212 555-1234"
  },
    {
    :type "fax",
    :number "646 555-4567"
  } ]
}
```
## Extensible Data Notation (edn)

<table>
<thead>
<tr>
<th>type</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>&quot;foo&quot;</td>
</tr>
<tr>
<td>character</td>
<td>\f</td>
</tr>
<tr>
<td>integer</td>
<td>42, 42N</td>
</tr>
<tr>
<td>floating point</td>
<td>3.14, 3.14M</td>
</tr>
<tr>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>symbol</td>
<td>foo, +</td>
</tr>
<tr>
<td>keyword</td>
<td>:foo, ::foo</td>
</tr>
</tbody>
</table>
## Extensible Data Notation (edn)

<table>
<thead>
<tr>
<th>type</th>
<th>properties</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>sequential</td>
<td>(1 2 3)</td>
</tr>
<tr>
<td>vector</td>
<td>sequential and random access</td>
<td>[1 2 3]</td>
</tr>
<tr>
<td>map</td>
<td>associative</td>
<td>{::a 100, ::b 90}</td>
</tr>
<tr>
<td>set</td>
<td>membership</td>
<td>#{::a ::b}</td>
</tr>
</tbody>
</table>
Extensible Data Notation: Clojure

semantics:

```
(fn call arg)

(println "Hello World")
```

structure:

```
(list symbol string)
```
Extensible Data Notation: Clojure

```clojure
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```
Extensible Data Notation: Clojure

(symbol symbol)

(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
Extensible Data Notation: Generic Extension

- #name edn-form
  - *name* describes the interpretation/domain of the element that follows
  - Recursively defined

- Built-in tags
  - #inst “rfc-3339-format”
    - Tagged element string in **RFC-3339 Format**
    - #inst “1985-04-12T23:20:50.52Z”
  - #uuid “canonical-uuid-string”
    - Tagged element is a UUID string
    - #uuid “f81d4fae-7dec-11d0-a765-00a0c91e6bf6”
Persistent Data Structures

- Immutable
- “Change” is by function application
- “Change” produces a new collection; structurally shared
  - Full-fidelity old version remains available
- Maintains performance guarantees
- Built upon linked lists and hash array mapped tries (HAMT(s))
Persistent Data Structures
Persistent Data Structures
Persistent Data Structures
## Persistent Data Structures

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Mutable, Transient</th>
<th>Immutable, Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing</td>
<td>difficult</td>
<td>trivial</td>
</tr>
<tr>
<td>Distribution</td>
<td>difficult</td>
<td>easy</td>
</tr>
<tr>
<td>Concurrent Access</td>
<td>difficult</td>
<td>trivial</td>
</tr>
<tr>
<td>Access Pattern</td>
<td>eager</td>
<td>eager or lazy</td>
</tr>
<tr>
<td>Caching</td>
<td>difficult</td>
<td>easy</td>
</tr>
<tr>
<td>Examples</td>
<td>Java, .Net Collections, Relational DBs, Place-Oriented Systems</td>
<td>Clojure, F# Collections, Datomic DB, Value-Oriented Systems</td>
</tr>
</tbody>
</table>
Persistent Data Structures

### Functions:

<table>
<thead>
<tr>
<th>Action</th>
<th>List</th>
<th>Vector</th>
<th>Map</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>list, list*</td>
<td>vector, vec</td>
<td>hash-map, sorted-map</td>
<td>set, hash-set, sorted-set</td>
</tr>
<tr>
<td>Examine</td>
<td>peek, pop, list?</td>
<td>get, nth, peek, vector?</td>
<td>get, contains?, find, keys, vals, map?</td>
<td>get, contains?</td>
</tr>
<tr>
<td>“Change”</td>
<td>conj</td>
<td>conj, assoc, subvec, replace</td>
<td>assoc, dissoc, merge, select-keys</td>
<td>conj, disj</td>
</tr>
</tbody>
</table>
Unified Succession Model

- **Separation of State and Identity**
  - Identities are managed references to immutable values
    - References refer to point-in-time value
  - Values aren’t updated in-place
  - Function application moves state forward in “time”
  - References see a succession of values
    - \((\text{change-state reference function args}^*)\)

- **Clojure provides reference types**
  - Synchronous
    - Var, Atom (uncoordinated)
    - Ref (coordinated; Uses STM)
  - Asynchronous
    - Agent
Unified Succession Model

Value

Given some value
Unified Succession Model

Given some function $f$, the value...
Unified Succession Model

Atomically apply the function to the value; “Atomic Succession”
Unified Succession Model

Which results in a new value, at a new point in time
Unified Succession Model
Unified Succession Model

A reference sees a succession of values
Unified Succession Model

Observers perceive identity; can see each value, can remember and record
Unified Succession Model

Observers do not coordinate
Unified Succession Model

\[(\text{def counter } (\text{atom } 0))\]
Unified Succession Model

\[
\text{atom} \quad 0 \quad \text{swap!} \quad 10
\]

\[
\text{(def counter (atom 0))}
\]

\[
\text{(swap! counter + 10)}
\]
 Unified Succession Model

(def counter (atom 0))
(swap! counter + 10)

atom 0 swap! 10

Atomic Succession

(def counter (atom 0))
(swap! counter + 10)
Unified Succession Model

(def counter (atom 0))
(swap! counter + 10)

Pure function
Unified Succession Model

today

Nov 8
Unified Succession Model

Identity: today

State:

Nov 8 Value ➔ function ➔ Nov 9 Value
Sequence Abstraction

- Clojure is a language programmed to interfaces/abstractions
  - Collections are interfaces, Java interfaces for interop, etc.

- Sequence interface unifies the foundation
  - Sequential interface
  - Used like iterators/generators, but immutable and persistent

- "It is better to have 100 functions operate on one data-structure abstraction than 10 functions on 10 data-structure abstractions."

- Clojure’s core is made up of functions of data-oriented interfaces/abstractions
  - Seqs work everywhere: collections, files/directories, XML, JSON, result sets, etc.
Sequence Abstraction

- **first / rest / cons**
  - (first [1 2 3 4])
    -> 1
  - (rest [1 2 3 4])
    -> (2 3 4)
  - (cons 0 [1 2 3 4])
    -> (0 1 2 3 4)

- **take / drop**
  - (take 2 [1 2 3 4])
    -> (1 2)
  - (drop 2 [1 2 3 4])
    -> (3 4)

- **Lazy, infinite**
  - (iterate inc 0)
    -> (0 1 2 3 4 5 ...)
  - (cycle [1 2 3])
    -> (1 2 3 1 2 3 1 2 3 ...)
  - (repeat :a)
    -> (:a :a :a :a ...)
  - (repeatedly (fn [ ] (rand-int 10) ) )
    -> (3 7 1 4 6 7 4 7 ...)
Sequence Abstraction

- **map / filter / reduce**
  - (range 10)
    -> (0 1 2 3 4 5 6 7 8 9)
  - (filter odd? (range 10))
    -> (1 3 5 7 9)
  - (map inc (range 10))
    -> (1 2 3 4 5 6 7 8 9 10)
  - (reduce + (range 10))
    -> 45

- **Fibonacci Sequence**
  - (def fibo
    (map first (iterate (fn [[a b]] [b (+ a b)]) [0 1])))
  - (take 7 fibo)
    -> (0 1 1 2 3 5 8)
  - (into [] (take 7 fibo))
    -> [0 1 1 2 3 5 8]
Sequence Abstraction

- What actors are in more than one movie, topping the box office charts?
  - Find the JSON input data of movies
  - Download it
  - Parse the JSON into a value
  - Walk the movies
  - Accumulating all cast members
  - Extract actor names
  - Get the frequencies
  - Sort by the highest frequency
Sequence Abstraction

- What actors are in more than one movie, topping the box office charts?

  (->> “http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies”
    slurp
    json/read-json
    :movies
    (mapcat :abridged_cast)
    (map :name)
    frequencies
    (sort-by val >))))
Reducers

```clojure
(ns ...
  (:require [clojure.core.reducers :as r]))

->> apples
  (filter :edible?)
  (map #(dissoc % :sticker))
  count)
```

Transducers

- Composable algorithmic transformations
  - Independent of/decoupled from their input and output sources

- A single “recipe” can be used many different contexts/processes
  - Collections, streams, channels, observables, etc.

- On-demand transformation characteristics
  - Decide between eager or lazy processing, per use (separate from the “recipe”)

- Same sequence API, without the source sequence
Transducers

- **map / filter**
  - (filter odd?) ;; returns a transducer that filters odd
  - (map inc) ;; returns a mapping transducer for incrementing

- **take / drop**
  - (take 5) ;; returns a transducer that will take the first 5 values
  - (drop 2) ;; returns a transducer that will drop the first 2 values

- **Composition is function composition**
  - (def recipe (comp (filter odd?)
    (map inc)
    (take 5)))
Protocols

● Named set of generic functions

● Provide a high-performance, dynamic polymorphism construct
  ○ Polymorphic on the type of the first argument

● Specification only; No implementation

● Open extension after definition
Protocols

(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
Protocols

(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))

(baz "hello")

java.lang.IllegalArgumentException: No implementation of method: :baz
of protocol: #'user/AProtocol
found for class: java.lang.String
Protocols

(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))

(extend-protocol AProtocol String
  (bar [a b] (str a b))
  (baz [a] (str "baz-" a)))

(baz "hello") => “baz-hello”
Protocols (and other forms of polymorphism)

<table>
<thead>
<tr>
<th>Closed for Extension</th>
<th>Open for Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch maps</td>
<td>Multiple dispatch / multimethods</td>
</tr>
<tr>
<td>Conditional dispatch</td>
<td>Protocols (type of first arg only)</td>
</tr>
<tr>
<td>Pattern-matching dispatch</td>
<td></td>
</tr>
</tbody>
</table>
Programming models are libraries

- “Program in data, not in text”
  - Program manipulation is data manipulation

- Extend the language using the language
  - Functions, macros, extensible reader (edn)

- Build the language up to your domain
  - What’s the ideal way to solve your exact problem?
  - You never need to wait for the language to evolve

- Examples
  - core.async
  - core.logic
Programming models are libraries: **core.async**

- Async programming using channels and CSP
- No bytecode rewriting, no Clojure modifications -- just a library
- Go blocks, IOC ‘threads’, parking, channels separate from buffers, etc.

```clojure
(let [messages (chan)]
  (put! messages "ping")
  (go (println (<! messages))))
```
Programming models are libraries: **core.logic**

- Logic programming as a library
- Prolog-like relational programming, constraint logic programming, and nominal logic programming
- Extensible to other forms of logic programming
- Sudoku solver, type inferencer, and more **examples**
Programming models are libraries: \texttt{core.logic}

\begin{verbatim}
(defrel rps winner defeats loser)

(fact rps :scissors :cut :paper)
(fact rps :paper :covers :rock)
...
(fact rps :rock :breaks :scissors)

(run* [verb]
  (fresh [winner]
    (rps winner verb :paper))))
\end{verbatim}
clojure.spec

● Docs are not enough

● Predicative specifications of data

● Values, maps, and sequences

● Validation, error reporting, parsing/destructuring, instrumentation, test data generation, property-based generative test generation
user=> (require '[clojure.spec :as s])
(s/def ::even? (s/and integer? even?))
(s/def ::odd? (s/and integer? odd?))
(s/def ::a integer?)
(s/def ::b integer?)
(s/def ::c integer?)
(def s (s/cat :forty-two #{42}
   :odds (s/+ ::odd?)
   :m (s/keys :req-un [:a ::b ::c])
   :oes (s/* (s/cat :o ::odd? :e ::even?))
   :ex (s/alt :odd ::odd? :even ::even?)))
user=> (s/conform s [42 11 13 15 {:a 1 :b 2 :c 3} 1 2 3 42 43 44 11])
{:forty-two 42,
 :odds [11 13 15],
 :m {:a 1, :b 2, :c 3},
 :oes [{:o 1, :e 2} {:o 3, :e 42} {:o 43, :e 44}],
 :ex {:odd 11}}
Clojure: Software Engineering

- Modern language, built for modern systems, to build modern systems
- First-class specification & instrumentation; Design-by-contract
- Robust testing spans unit, generative property-based, and simulation testing
- Architecturally evident (namespaces), low cognitive load
- Ecosystems and reach
Clojure: Community and Ecosystems

- Mailing list / Slack / IRC / Events - [https://clojure.org/community](https://clojure.org/community)
- Community-driven examples and docs - [https://clojuredocs.org](https://clojuredocs.org)
- IDE/Editor support, all with interactive development support
- Project tooling: Maven, Gradle, Boot, Leiningen, etc
- Professional Services, Support, Training from Cognitect - [http://cognitect.com/](http://cognitect.com/)
- Robust Web Services with Pedestal - [http://pedestal.io/](http://pedestal.io/)
- Rapid Microservices with Vase - [https://github.com/cognitect-labs/vase](https://github.com/cognitect-labs/vase)
  - Microservices expressed as data/edn
Clojure applied: Walmart “Savings Catcher”

- Process and integrate every purchase
  - 5000+ physical stores
  - Online and mobile purchases
  - Globally distributed system and data

- Savings Catcher, Vudu Instawatch, Black Friday 1-Hour Guarantee

- 8 Developers
Clojure applied: Boeing 737 MAX Diagnostics

- Diagnostic system similar to car’s “Check Engine” light
- Hundreds of sensors, streams of data, constant calculation
  - Value validations
  - Interdependent rules evaluation
  - Over 6,000 possible codes
  - Only 34,000 lines of Clojure
- Fully integrated into the flight-deck and ONS system (laptops/tablets/etc)
- “Clojure is a relatively new software language that allowed us to write rules and code capable of handling massive amounts of data under significant hardware limitations”
- Significant cost/time savings; Improved error detection and accuracy
Clojure applied: DRW Trading

- Already on the JVM; Existing systems in Java
- Needed “speed”
  - Execution performance important
  - Time-to-value-delivered
- Interactive-development increased productivity
  - Production debugging
  - Exploratory adaptations
- Domain dominated by data
  - Data-oriented abstractions simplified solutions
  - Whole classes/libraries turned into single Clojure functions