An short intro to Clojure for Rubyists

Trifork A/S, Headquarters
Margrethepladsen 4,
DK-8000 Århus C,
Denmark
info@trifork.com
http://www.trifork.com

Aarhus Ruby Brigade
December 6th, 2010,

What exactly are the reasons we like Ruby so much?

Clojure a Ruby Perspective

- Language has the following properties:
  - Dynamically typed
  - Reflexive
  - Consistent (alá Principle of least surprise)
  - Supports Meta-programming and DSLs
  - Supports easy exploration and experimentation (interactive)
  - Open to “extension” (even String and other 'base' types).
  - Blocks/Closures are first-class citizens
  - Many abstractions (like sequentiality) and a big standard library that work on those abstractions
    - These work with most language data structures, (e.g., map, reduce, “foreach” on “arrays”, “hashmaps”,...)
  - Literal syntax (maps, arrays, symbols,regexps,...)
  - And (much) more...
- Sounds familiar?

What is Clojure (in one slide)

- **Functional dynamic language**
  - Persistent data structures, pure functions
- **A unique programming & concurrency model**
  - State management constructs: var, ref, atom, agent
- **On-the-fly & AOT compilation: JVM bytecode**
  - Deep two-way Java interop.; embraces host
- **A new, different LISP family member**
  - Meta programming, closures

- Sound unfamiliar? No, Clojure is not Ruby, but they share many ideas and values.

JVM != Java

JVM is a powerful platform for many dynamic languages: Clojure, JRuby, Erjang, JPython, ...

HotSpot

Garbage Collection

Concurrency features & Memory model

Libraries and Tools

Polyglot “monoplat” Interop

Working hypothesis

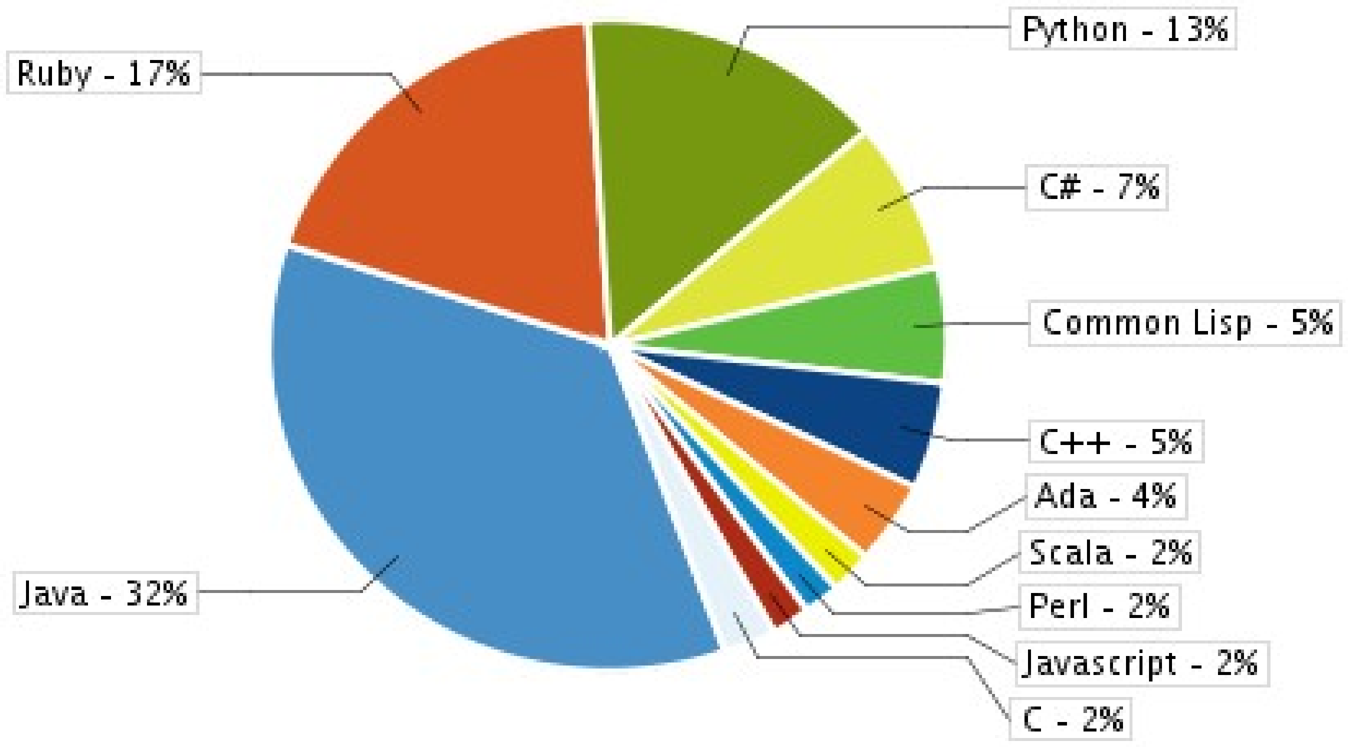
Ruby users will automatically like most features of Clojure

```
krukow:~$ man ruby
...
If you want a language for easy object-oriented programming, or you
don't like the Perl ugliness, or you do like the concept of LISP,
 but don't like too much parentheses,
Ruby may be the language of your choice.
```

Who is picking up Clojure?

http://cemerick.com/2010/06/07/results-from-the-state-of-clojure-summer-2010-survey/

What language did you use just prior to adopting Clojure?



Agenda

- Immutability and functional programming in Clojure
- Briefly on syntax,meta programming, extensibility
- Demos: interactivity & dynamism
  - Host interop (Jacob will also speak about interop)
  - Sequence library and maps
  - Macros (DSLs, meta programming)

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”`

Composite Data Types

- 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

Persistent Data Structures

- In Clojure, generally, all data structures and objects are *immutable*
- There are efficient operations operations for creating *variants* of a data structure, for example:
  - If **`M`** is a clojure map, then **`assoc(M,k,v)`** is a clojure map which is like **`M`** except that it maps key **`k`** to value **`v`**
  - (within 1-4x their mutable counterparts, or faster :)
- Further: they are *persistent* meaning that the operations are non-destructive, e.g.,
  - Both **`M`** and **`assoc(M,k,v)`** are usable and preserve their performance guarantees.

Why immutability?

- Immutability removes a lot of complexity
  - You can safely pass an immutable object to unknown code without "defensive copying"
  - You can store an immutable object and come back later to look at it
  - Enables pure functions which are easier to understand, test and write
  - Enables compiler optimizations
  - Can be shared among threads without synchronization
- In contrast graphs of mutable objects are

  "the new spaghetti code"

NO!

... that doesn't entail copying the entire structure(!)

See my blog for explanation:

http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/

http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice/

http://blog.higher-order.net/2010/08/16/assoc-and-clojures-persistenthashmap-part-ii/

Also

http://blog.higher-order.net/2010/06/11/clj-ds-clojures-persistent-data-structures-for-java/

Sequence library and pure functions

Sequences: example of programming to abstractions

- Lift the first/rest abstraction off of concrete lists
- Function seq
  - (seq coll) gives **nil** if empty otherwise a seq on coll
  - **first**, calls seq on arg if not already a seq
    - returns first item or **nil**
  - **rest**, calls seq on arg if not already a seq
    - returns the next seq, if any, else **nil**
- Most library functions are fully lazy
- Vast library works on: all Clojure DS, Java: Strings, arrays, collections, iterables, your own types
  - All *pure functions*

Sequence library examples

```
drop 2, [1, 2, 3, 4, 5] → (3, 4, 5)

cycle [1, 2, 3] → (1, 2, 3, 1, 2, 3, …)

partition 3, [1, 2, 3, 4, 5, 6] → ((1 2 3) (4 5 6))

interpose \, "Fred" → (\F \, \r \, \e \, \d)

map inc, [1, 2, 3] → (2, 3, 4)

reduce +,
   range 100 → 4950
```

Examples: Map functions

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

m :b → 2      (maps are functions)

:b m → 2    (keywords are functions)

keys m → (:a, :b, :c)

assoc m, :d 4 → {:a 1, :b 2, :c 3, :d 4} ;;m is unchanged

dissoc m, :a → {:b 2, :c 3} ;;m is unchanged

map :name [{:name "Fred"}, {:name "Ethel"}]
                   → ("Fred", "Ethel")

merge-with +, m, {:a 2, :b 3} → {:a 3, :b 5, :c 3}
```

TRIFORK.

Records: another kind of object

- Records aren't like objects in traditional OO languages (think Java), but there are similarities
  - Represent “domain types”
  - Store data in fields; can have associated “methods”
  - Can implement host interfaces
  - Fast field access (via :keyword)
- But there are also differences with traditional objects
  - Records are immutable ('setters' create new records)
  - Automatic impl. of hashCode, equals, toString
  - Extensible and map-like behaviour: implementents interfaces clojure.lang.IPersistentMap and java.util.Map
  - No inheritance
  - Methods & polymorphism only via interfaces and protocols (more on that)

If it were Java a record would be similar to this

```java
public class PersonRecord implements IPersister
{
        public final String firstName;
        public final Integer age;
        public final AddressRecord address;
        public PersonRecord(String firstName,
```

```
                                  )bject val) {
                                  s one
                return null;//
        }
        //... more methods...
}
```

```ruby
class PersonRecord
  include Enumerable
  include FieldEnum
  attr_accessor :name, :age, :address
end
```

```ruby
module FieldEnum
  def each
    instance_variables.each do |v|
      yield(instance_variable_get v)
    end
    nil
  end
end
```

```clojure
defrecord Person,
       [f rstname, age,  address]
```

Map-like nature of Records

```
defrecord Person
    [f rstname, age,  address]
```

```
def p
  Person. "Karl", 42, nil
```

- The map-like nature of records means
  - Records can be extended as a map

```
assoc p, :lastname, "Krukow"   ──►   #:Person{:f rstname "Karl",
                                              :age 42,
                                              :address nil,
                                              :lastname "Krukow"}
```

  - In general, any function that works on maps works on records

```
keys p          ──►   (:f rstname, :age, :lastname)

second,
    vals p      ──►   42
```

Extensibility

- There is a concept of protocols which are a grouping of polymorphic functions.
  - for example consider an “Indexed” protocol
  - Extended to Java native type String

```
defprotocol Indexed,
  at [this, i] "return el at ith pos"
```

```
extend-protocol Indexed,
   String,
     at [str i],
       . str charAt i
```

```
at "Karl" 2

;result is → \r
```

- Can be done at any point in the program

TRIFORK.

Confession...

I've not presented Clojure syntax completely accurately...

Clojure syntax

- Clojure compiler is defined in terms of data structures, not text.
- There are textual representations of all the data structures and objects (),[],{},#{},...
  - There are some parens – but not too many, actually

```
(      (    ))

(
  (   [  (           )]
  (            (   )))
```

```
(reduce + (range 100))
```

- Clojure is at least as compact as Ruby/Python...
  - (See Peter Norvig's spellchecker in Clojure)
  - http://norvig.com/spell-correct.html

Syntactic weight: Ruby 1.9 vs Clojure

Note semantics is vastly different although syntax is similar

```ruby
m = {:name, "Karl", :age, 42}
# => {:age=>42, :name=>"Karl"}

m[:name]
# => "Karl"


class PersonRecord
  attr_accessor :name, :age
end

p = PersonRecord.new
p.name = "Karl"; p.age=42

[p,p,p].map &:name
#=> ["Karl", "Karl", "Karl"]


[p,p,p].map(&:age).reduce(&:+)
#=> 126
```

```clojure
(def m {:name "Karl", :age 42})

(m :name)
(:name m)






(defrecord Person[name age])
(def p (Person. "Karl" 42))



(map :name [p p p])
;;=> ("Karl" "Karl" "Karl")

(reduce + (map :age [p p p]))
;;=> 126

(->> [p p p] (map :age)(reduce +))
;;=> 126
```

TRIFORK.

Macros: Extending the language

- Clojure supports macros as a mechanism for extending the language itself.
- A macro is simply a function that takes a program as input and produces a program as output (program = data structure)
- Note: a macro call doesn't evaluate its args
- For example: Ruby has **unless**

```
(defmacro unless [test then]
              `(if (not ~test) ~then))

(unless (= 2 3)
       "Clojure")
;=> “Clojure”
```

```
(ruby-style

       "Clojure" unless (= 2 3))
;;=> “Clojure”

;;where
(defmacro ruby-style [exp pred clause]
              `(~pred ~clause ~exp))
```

Ruby → Clojure?

- Why would you consider Clojure instead of Ruby?
  - You want to keep a dynamic, interactive and meta-programming environment
  - But any one of the following are true
    - You need more Speed
    - Your problem is Concurrent
    - You problem is too Complex for mutable objects :)
    - You need more meta programming and DSLs
    - You want to interoperate with JVM languages

Please note that there is so much more to Clojure than what I've presented

So do checkout the references, particularly the videos of Rich Hickey

http://clojure.blip.tv/

References

- DCUG: www.clojure.dk
- http://clojure.org/
- http://clojure.blip.tv/
- Stuart Halloway: Programming Clojure
  - intro
- Chris Houser, Michael Fogus: The Joy of Clojure
  - Deeper
- Rich Hickey
  - Are we there yet?
    http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey
    Value, state, identity
    http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey