Clojure

Meta

- Talk-structure and many slides and stolen from Rich Hickey ;-)

  see http://clojure.org/

- Thanks to Azul Systems for letting us play with their cool tech.
  - See http://www.azulsystems.com/
- Thanks to Cliff Click for helping out, and letting me use his program to reproduce his Clojure experiments from JavaOne 2009.
  - http://blogs.azulsystems.com/cliff/

Clojure in one slide

- **Functional language**
  - Immutabiltity via persistent data structures
- **A new, very general LISP family member**
  - Dynamic, syntactic abstraction
- **On-the-fly compilation to JVM bytecode**
  - Deep two-way Java interop.; idiomatic using Java
- **Focus on and support for Concurrency**
  - A unique concurrent programming model

Why Clojure?

- Designed to embrace the Host: JVM(s)
  - Contrast to ”ports”: e.g., JRuby, Jython
  - Other natives: Groovy, Scala
- Expressive, elegant, extensible
- Good performance
  - e.g. Cliff Click: http://www.azulsystems.com/events/javaone_2009/session/2009_J1_JVMLang.pdf
- Wrapper-free Java access; use Clojure in Java
- Unique lock-free concurrency model

Agenda

- **Introduction to Clojure**
  - Way to much to cover in one hour!
- **In depth with persistent data structures**
  - ”secret sauce” of Clojure (according to Hickey :-)
- **Concurrent TSP solution using Azul Box**
  - Azul Systems, ”Vega-3”, 864 core, 364 GB Ram!!!

A dynamic language

- Dynamically typed
  - Flexibility, productivity, concision
- Interactive development
  - Read-Evaluate-Print-Loop (REPL)
  - Introspection
- Although: No extensible base-classes like Ruby
  - Shares types with Java, e.g., String

A *new* LISP

- Core is extremely simple and small
- Code-as-data
  - Compiler defined in terms of data structures not text
- Macros, i.e., syntactic abstraction
  - User-defined functions extending compiler (DSLs!)
- *NEW:* very functional; concurrency semantics
- *NEW:* Programs composed of all types of DS
- *NEW:* Abstract sequences generalize lists

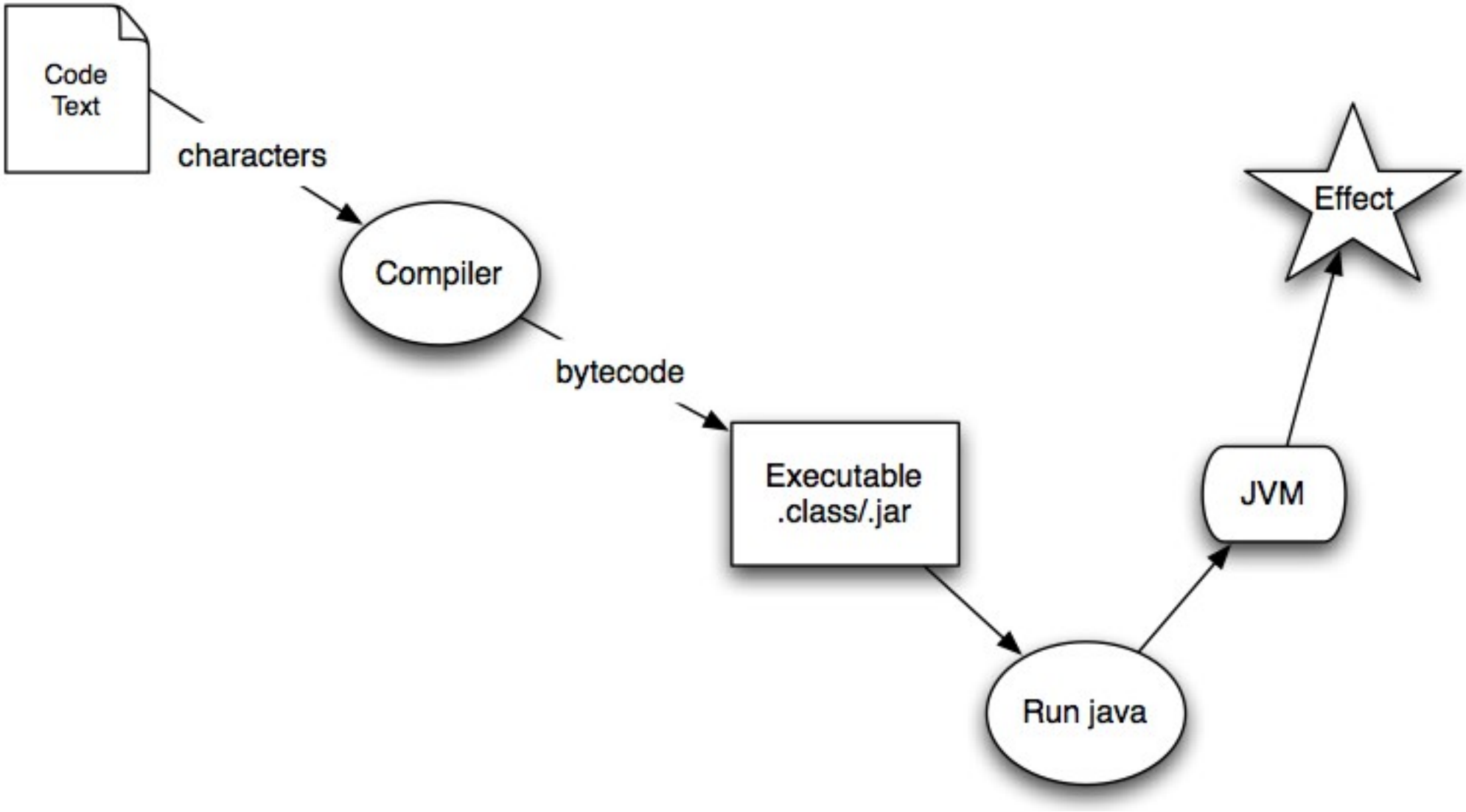
Atomic Data Types*

- 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"`

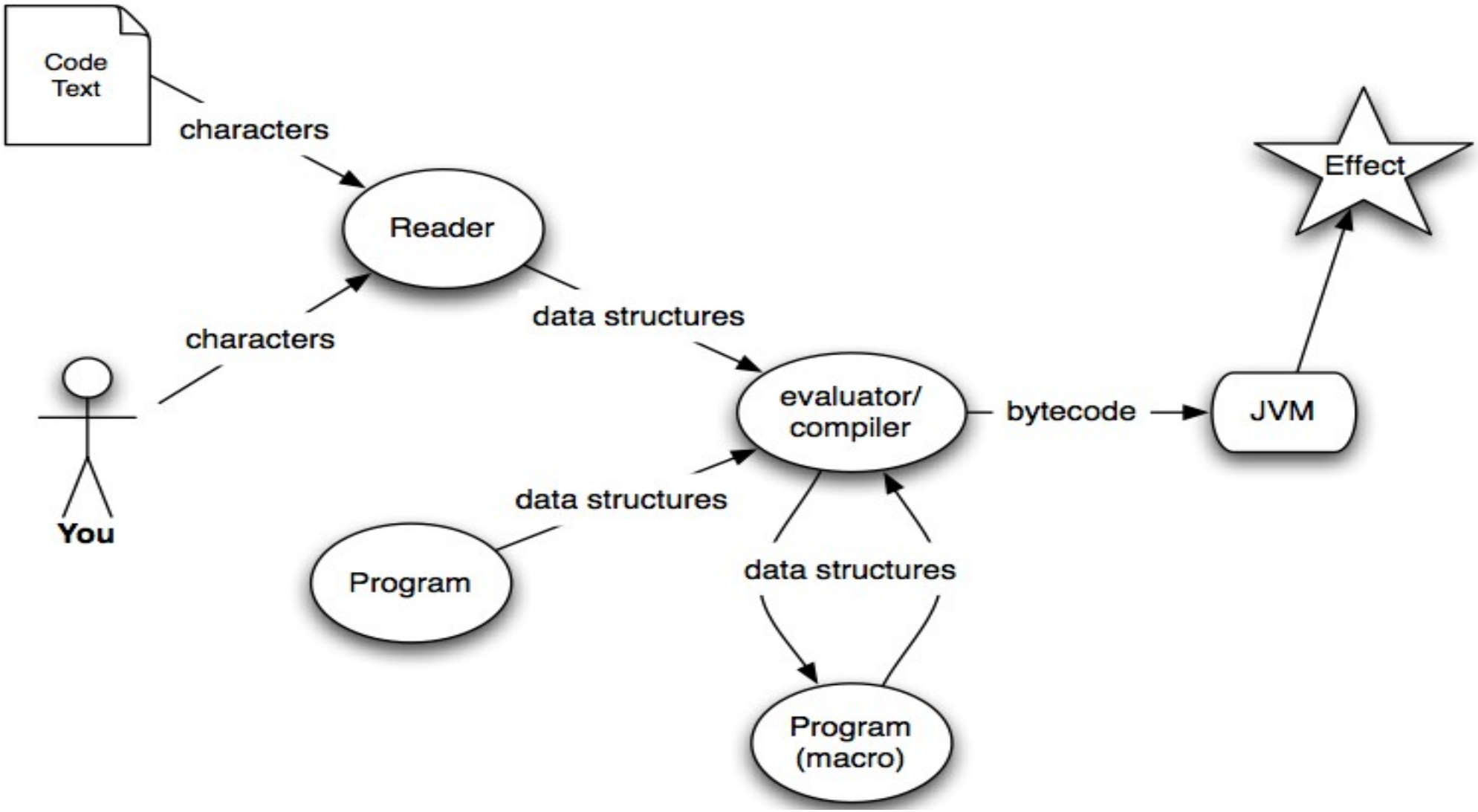
Data Structure Literals*

- 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

Traditional evaluation model*



Clojure evaluation model*



Syntax

- That's it :-)
  - Homoiconicity
- Of form: parenthesized list with operator first
  - `(op a1 a2 … )`
- 'op' is either
  - Special op, e.g. def, .x (?), ref... (12 (14) total)
  - Macro, e.g. doto, with-open, user-defined
  - Function/callable-exps, e.g. list, conj, user-defined
- 'op' determines what the compiler does

More on Macros

- Defined similarly to a function, defmacro
- When called, args as passed unevaluated!
- Difference with functions?
  - There are cases where you want to control evaluation of arguments
    - Language constructs, DSLs
  - Want more flexible syntax

```
(or exp1 exp2)
  ;=>
(let [x exp1]
    (if x x exp2))
```

- Sometimes you only need H-O functions

Macro Example

```
import java.io.*;
import java.util.*;
public class WithOpenStreamShort {
 public static void main(String[] args) throws IOException {
    InputStream st = null;
    try {
        st = String.class.getResourceAsStream("/x.properties");
        Properties p = new Properties();
        p.load(st);
        System.out.println(new HashMap(p));
    } finally {
        if (st != null) {
            st.close();
        }
    }
 }
}
```

```
(println
  (with-open [s (.getResourceAsStream String "/x.properties")]
       (into {} (doto (java.util.Properties.) (.load s)))))
```

Example: Defining macros

```
(defmacro with-open
   "modified version of with-open"
  [bindings body]
   `(let ~bindings
      (try ~body
         (finally
            (if ~(bindings 0) (. ~(bindings 0) close))))))
```

- Syntax-quote: `
- gen-sym: x#
- Unquote: ~
- `macroexpand` is your friend

Sequences

- 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

Sequence Library*

```
(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])
-> ([:a 1] [:b 2] [:c 3] [:d 4] [:e 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}}
```

Java Interoperability

- At the level of types
- Core library
- Language level (syntax!!, wrapper-free use)
- At level of instances (`proxy` and `new`)
- At level of classess (`gen-class, compile`)
- Extensible by design

Language Level

- 'Syntax' (built-ins and core macros)
  - `new`, `try`, `set!`, `.`
  - `..`, `doto`, `with-open`, `instance?`, ...

- Wrapper-free access (`import` or `ns`)

```
(import '(java.util StringTokenizer))
   (.nextToken (new StringTokenizer "r,i,c,h" ","))
```

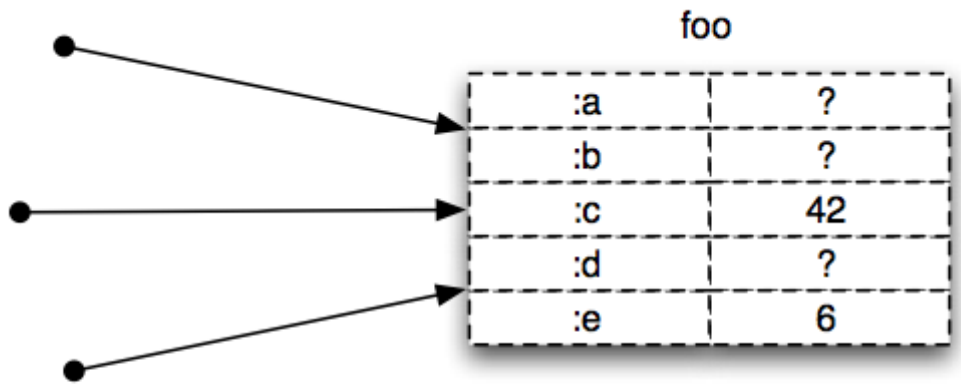
Clojure Concurrency Model

- *Functional Programming:*

  ”The philosophy behind Clojure is that most parts of most programs should be functional, and that programs that are more functional are more robust.”

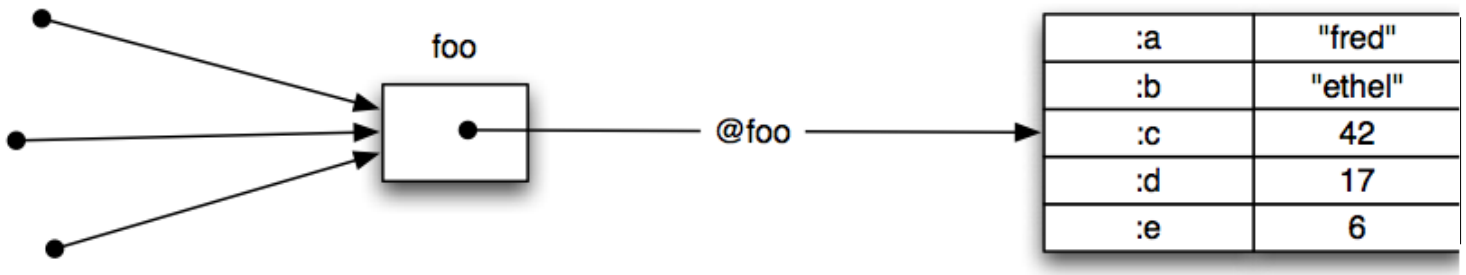
- *Indirect references* to immutable data
  - Explicit reference types (inspired by SML's `ref`)
  - ***Only*** *references* mutate (not data)
    - Mutation, is system-controlled, always atomic
  - Efficient immutability with persistent data structures
  - New state is computed as *functions* of the old state

Traditional OO approach:
Direct References to Mutable Objects*



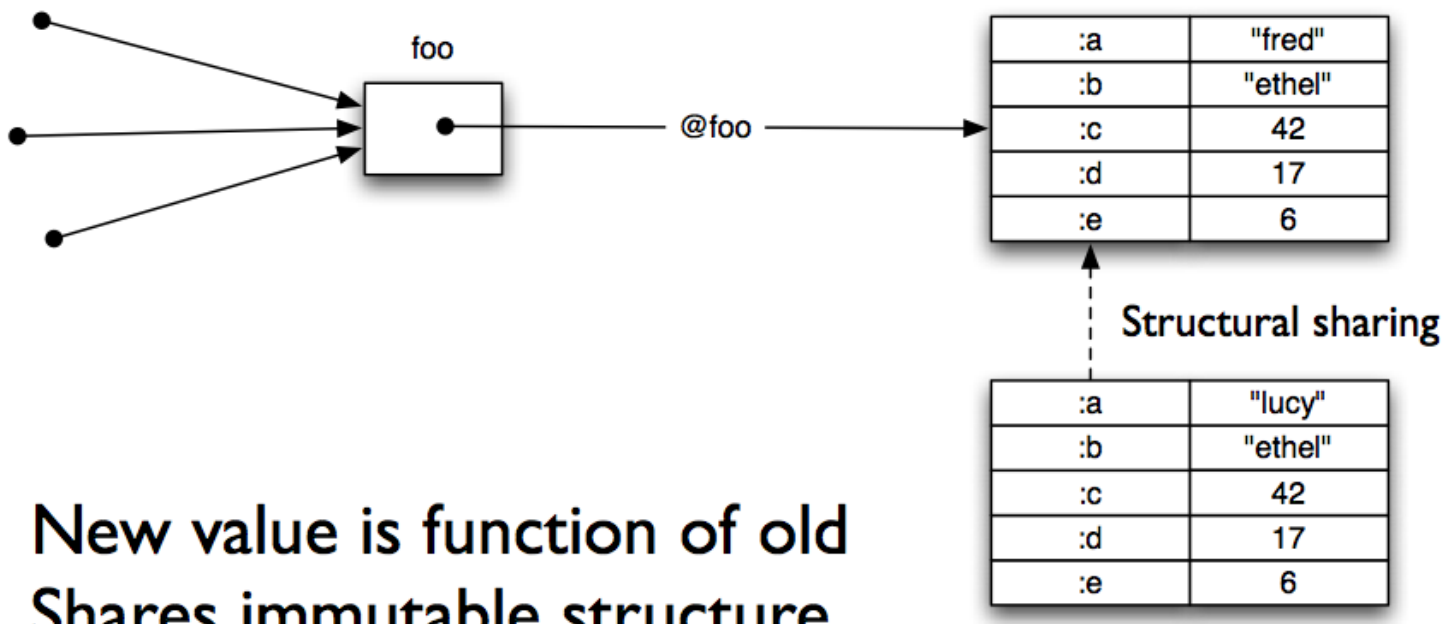
- Unifies identity and value
- Anything can change at any time
- Consistency is a user problem

Clojure Approach:
Indirect References to Immutable Objects*



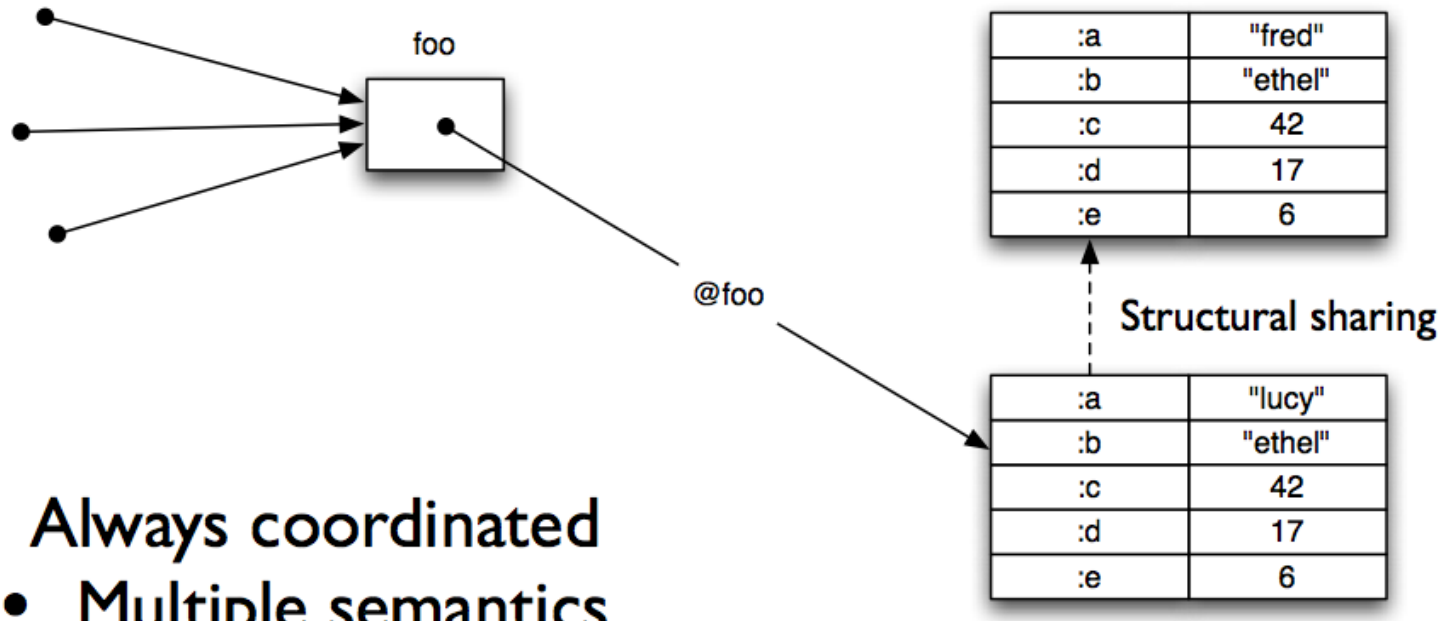
- Separates identity and value
  - Obtaining value requires explicit dereference
- Values can never change
  - Never an inconsistent value

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

Clojure References*

- Only references mutate: in a controlled way
- 4 types of references, all with concurrency semantics:
  - Vars: shared root binding, isolate changes in thread
  - Refs: synchronous, coordinated
  - Atoms: synchronous, independent
  - Agents: asynchronous, independent
- deref or reader-macro @ to get value (not vars)
- Different mutator functions for each type

Refs and Transactions*

- Software transactional memory system (STM)
- Refs can only be changed within a transaction
- All changes are Atomic, Consistent 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
  - User 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

Almost...
STM can be
configured
to control
history-size

Summary

- Clojure is written for JVMs
  - Excellent Java interop; leverages JVM tech.
- Concurrency focus and support
  - refs, agents, atoms, vars; STM
- Functional programming
  - Immutability via Persistent Data Structures
- Lisp
  - Syntactic abstraction, dynamic
- More! Multimethods, ad-hoc hierarchies, meta-data, transients, chunked-seqs, AOT compilation, watchers..

References

- http://clojure.org has excellent documentation
- www.clojure.dk Danish Clojure Users' Group – signup, meet-up; read, write about Clojure
- Hours of video: http://clojure.blip.tv/
  - Rich Hickey is a great speaker!
- JAOO Aarhus 2009, Oct. 5-7th
  - http://jaoo.dk/aarhus-2009/speaker/Rich+Hickey
  - Introducing Clojure, The Clojure Concurrency Story, Concurrency Expert Panel
- Stuart Halloway, Programming Clojure (www.pragprog.com)
- Mark Volkmann, STM article
  - http://java.ociweb.com/mark/stm/article.html
- I blog about Clojure (and other tech stuff :-)
  - http://blog.higher-order.net