Die Sprache Clojure

Benjamin Teuber

Kunterbuntes Seminar
Department Informatik
Universität Hamburg

18. Mai 2010

Intro

Clojure - Überblick

- Funktional
  - aber nicht ausschließlich
- Dynamisch Typisiert
  - erlaubt Type Hints
- Basiert auf JVM
  - alternativ CLR
- Ein modernes Lisp
  - mehr Syntax
  - weniger Klammern
  - gleiche Mächtigkeit

Designziele

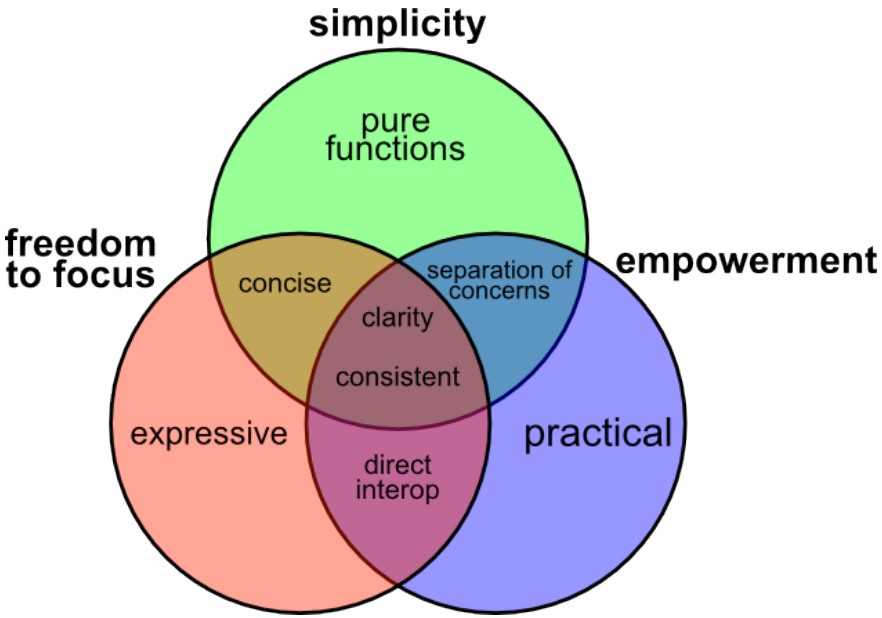


Abbildung: Clojure-Designziele
(Quelle: “The Joy of Clojure”)

- Allgemein
  - Einfachheit
  - Ausdrucksstärke
  - Praktikabilität
- Konkret
  - Kaum Syntax
  - Trennung von Konzepten
  - exzellente Java-Interop
  - Erweiterbarkeit

Basics

Atomare Werte

- Zahlen
  - `1, 3.14, 3/4`
- Booleans
  - `true, false`
- Strings
  - `"Hallo, Welt!"`
- Keywords
  - `:black, :name`
  - Stehen für sich selbst
- Symbole
  - `x, my-name, finished?`
  - Stehen für etwas anderes
- `nil`
  - Steht für “nichts”

Datenstrukturen

- Listen
  - `(a b c)`
  - In der Regel für “Befehle” verwendet
- Vektoren
  - `[1 2 3]`
  - Normalerweise für Daten verwendet
- Maps
  - `{:a 1, :b 2, :c 3}`
- Sets
  - `#{1 42 :foo}`

Syntax

- Habt ihr bereits gesehen!
- Clojure ist homoikonisch
  - Code ist selbst in Datenstrukturen der Sprache repräsentiert

  $\Rightarrow$ Meta-Programmierung wird einfach!
- Aber: Semantik steckt in Auswertungsregeln

Beispiel: Funktionsdefinition

```
(defn my-add
  "Adds two numbers"
  [x y]
  (+ x y))
```

Auswertung

- Fast alles evaluiert zu sich selbst - bis auf ...
  - Symbole: Lookup des Wertes
  - Listen: Erstes Element wird als Befehl, alle weiteren als Elemente interpretiert
  - Evaluierung kann durch quote unterbunden werden
- Die Funktion `eval` liefert einen vollständigen Clojure-Interpreter

Der REPL

- Read-Eval-Print-Loop
  - Read: Umwandeln von Quelltext in Clojure-Datenstrukturen
  - Eval: Auswertung
  - Print: Ausgabe im Terminal
- Ernöglicht interaktive Entwicklung
  - Algorithmen entwickeln
  - Mit Libraries spielen
  - Testen

Funktionale Programmierung

Das Problem imperativer Sprachen

Python

```
x = [5]
process(x)
x[0] = x[0] + 1
```

- Was ist x am Ende?
- Semantik ist nicht-lokal
  - $\Rightarrow$ Systemkomplexität!
    - Hilft Kapselung?
- Nebenläufigkeit?
  - $\Rightarrow$ "Locking-Albtraum"

Zitat

"Mutable stateful objects are the new spaghetti code"
– Rich Hickey

Funktionale Programmierung

- Pure Funktionen
  - Keine Nebeneffekte
  - Ausgabe hängt ausschließlich von den Eingaben ab

  $\Rightarrow$ Referentielle Transparenz
  - Vereinfacht Testen etc.
- Clojure
  - Pure Funktionen normal
  - Kein Zwang (vgl. Haskell)

Funktionen höherer Ordnung

- Funktionen selbst sind Werte
  - Können als Ein- oder Ausgaben vorkommen
  - Anonyme Funktionen sinnvoll
- z.B. praktisch für Stream-Verarbeitung

```
(def digits [0 1 2 3 4 5 6 7 8 9]

(filter odd? digits)

(map inc digits)

(take-while (fn [x] (< x 4)) digits)
```

Wahrheit in Clojure

- Logisch falsch: `nil` und `false`
- Logisch wahr: alles andere
- Ungewohnt, aber praktisch
- `(or nil 42)`

Wertsemantik

- Werte sind unveränderlich
- Alle bisherigen Datentypen sind Werte (auch Collections!!!)
- Gleichheit von Werten über Struktur
- `(= [1 2 3] [1 2 3])`
- Vergleiche Java
  - `equals` unsauber für veränderliche Objekte
  - Mal gleich, mal ungleich?
  - Strings

Collections

Collections

- Clojure-Collections sind persistent
  - Achtung: nicht im Sinne von Datenbanken!
  - Operationen geben “neues Objekt” zurück
  - Altes bleibt intakt
- Performanz
  - Komplette Kopie zu langsam
  - Idee: Structural Sharing
  - Implementation als Baumstrukturen
  - $O(log_{32} n)$, d.h. praktisch konstant!

Collections (2)

- "Ändern" in verschachtelten Collections
  - `(assoc-in {:a [10 11 12]} [:a 0] 42)`
  - `(update-in {:a [10 11 12]} [:a 0] + 5)`
- Collections als Funktionen
  - `([1 2 3] 0)`
  - `({:a 1, :b 2} :b)`
- Keywords als Funktionen
  - `(:b {:a 1, :b 2}`

Destructuring

- Wie Pattern Matching (aber weniger mächtig)
- Erlaubt zerlegen von Collections z.B. in Parametern

```
(defn foo [x [a b c] z]
  (prn x a b c z))

(foo 42 [1 2 3] :bar)
```

Sequence-Abstraktion

- Analog zu Listen alter Lisps
  - Erstes Element:
    `(first liste)`
  - Rest-Liste:
    `(rest liste)`
  - Element hinzufügen:
    `(cons elt liste)`
- Modernisierung: Interface
  - vgl. `Iterable`
  - Bereits für "alles" implementiert

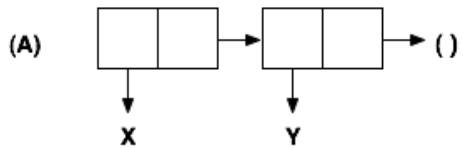


Abbildung: Alte Lisp-Listen
(Quelle: "The Joy of Clojure")

Lazy Sequences

- Lazyness: Verschieben von Arbeit bis Ergebnis benötigt
  - vergleiche Haskell
- Lazy Sequences
  - Können unendlich sein
  - Können große Datenmengen verwalten
  - Erlauben generischeren Code

$\Rightarrow$ Viele Sequence-Funktionen sind lazy

Beispiel: Fibonacci

```
(defn calc-fibs [x y]
  (lazy-seq
    (cons x (calc-fibs y (+ x y)))))

(def fibs (calc-fibs 1 1))
```

Java-Interop

Java-Interop allgemein

- von Java nach Clojure
  - Jede Clojure-Funktion implementiert Runnable
  - Jede Clojure-Collection implementiert Iterable
  - Jeder Clojure-Namespace wird eine Klasse
  - Clojure kann Klassen und Interfaces definieren
- von Clojure nach Java
  - Iterable, Strings, Arrays etc. sind Sequences
  - Klassen und Interfaces können geerbt werden

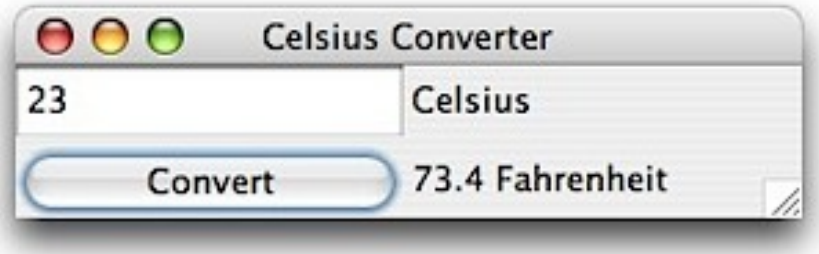
Java-Operatoren

- new
  - `(new java.io.File "foo")`
  - kürzer `(java.io.File. "foo")`
- Methoden
  - `(. person toString)`
  - kürzer `(.toString person)`
- Syntaktischer Zucker
  - `(. (. person getAdress) toString)`

  $\Rightarrow$ `(.. person getAdress toString)`
  - `(let [p (Person. "foobert")] (.setAge p 25) p)`

  $\Rightarrow$ `(doto (Person. "foobert") (.setAge 25))`

Beispiel: Swing-Anwendung



Beispiel: Swing-Anwendung (2)

```
(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener)
        '(java.awt GridLayout))
(defn celsius []
  (let [frame             (JFrame. "Celsius Converter")
        temp-text         (JTextField.)
        celsius-label     (JLabel. "Celsius")
        convert-button    (JButton. "Convert")
        fahrenheit-label (JLabel. "Fahrenheit")]
```

Beispiel: Swing-Anwendung (3)

```
(.addActionListener convert-button
  (proxy [ActionListener] []
    (actionPerformed [evt]
      (let [c (Double/parseDouble (.getText temp-text))]
        (.setText fahrenheit-label
           (str (+ 32 (* 1.8 c)) " Fahrenheit"))))))
(doto frame
  (.setLayout (GridLayout. 2 2 3 3))
  (.add temp-text)
  (.add celsius-label)
  (.add convert-button)
  (.add fahrenheit-label)
  (.setSize 300 80)
  (.setVisible true))))
```

State

OOP vs. Clojure-Sichtweise

- OOP-Sicht
  - veränderliche Werte
  - Alte Versionen zerstört
  - Probleme bei nebenläufigem Zugriff
- Clojure-Sicht
  - Veränderliche Referenz auf unveränderliche Werte
  - Alte Versionen intakt
  - “To excel at mutablility, you have to be good at immutability”

Abbildung: OOP-Sicht

Abbildung: Clojure-Sicht (Quelle: “The Joy of Clojure”)

Referenzarten

- 4 verschiedene Arten von Referenzen
  - Verschiedene Nebenläufigkeitssemantiken
  - Gleiche Schnittstellen
- Refs
  - Software Transactional Memory
  - Synchroner, koordinierter Zugriff
- Agents
  - Asynchroner, unkoordinierter Zugriff
- Atoms
  - Synchroner, unkoordinierter Zugriff
- Vars
  - Thread-lokal

Zugriff auf Refs

- Lesen
  - `@my-ref`
- Schreiben
  - `(ref-set my-ref)`
- Ändern
  - `(alter my-ref f extra-arg1 ...)`
- Schreibzugriff nur in Transaktion möglich
  - `(dosync ...)`

Beispiel: Bankkonten

- Atomare Überweisung
- Schwierig in Java

```
(def my-account   (ref 100))
(def your-account (ref   0))

(defn transfer [from to amount]
  (when (>= @from amount)
    (dosync
      (alter from - amount)
      (alter to   + amount))))

(transfer my-account your-account 50)
```

Fortgeschrittenes

Makros

- “Compiler-Plugins”
- erlauben echte Erweiterung der Sprache
- Idee: Funktionen, die Code als Ein- und Ausgaben bekommen
- viele Clojure-Befehle sind als Makros definiert

Aus clojure.core

```
(defmacro when
  "Evaluates test. If logical true,
   evaluates body in an implicit do."
  [test & body]
  (list 'if test (cons 'do body)))
```

Multimethods

- Idee: Verallgemeinerung von OO-Konzepten
  - Dispatch-Funktion
  - Hierarchien

```
(defmulti arg-count-dispatch (fn [& args] (count args)))

(defmethod arg-count-dispatch 1 [x]   "One Arg")
(defmethod arg-count-dispatch 2 [x y] "Two Args")
```

Protocols

- Multimethods allgemein, aber langsam
- 90% der Fälle single-dispatch
- Host-Sprache kann besser ausgenutzt werden

$\Rightarrow$ Protocols

- Interfaces++
- können nachträglich implementiert werden
  - ohne Wrapper
  - Namespace-abhängig

  $\Rightarrow$ keine Probleme mit externem Code
- keine Vermischung von Spezifikation und Hierarchie

Literatur

- Halloway: Programming Clojure (hier in der Bib)
- Fogus/Houser: The Joy of Clojure (bisher nur als Preview)
- Die Clojure-Webseite