CLOJURE

Georg Schmidhammer

Eigenschaften

- Funktionale Sprache (jedoch nicht rein funktional)
- Lisp Dialekt
- Beruht auf der JVM
- Nicht objekt-orientiert
- Auf Nebenläufigkeit optimiert

Entwicklungsgrund

- Rich Hickey suchte eine Sprache die:
  - Ein Lisp Dialekt ist
  - Auf einer weit verbreiteten Plattform beruht
  - Auf Nebenläufigkeit optimiert ist
- Fand keine Sprache die seinen Wünschen entsprach
- Entwickelte daraufhin Clojure

Geschichte

- Entwickelt von Rich Hickey
- Arbeitete vorher an ähnlichen Projekt, basierend auf .NET Framework
- Für Clojure 2 Jahre Arbeit
- 2007 in der Common Lisp Community veröffentlicht

Warum diese Eigenschaften?

- Lisp
  - Kleine Syntax
  - Code wird wie Daten behandelt
  - Sehr mächtig
- Funktional
  - Unveränderbare Variablen
  - Erleichtern Nebenläufigkeit
- JVM
  - Viele Bibliotheken
  - Kein Speichermanagement
- Nebenläufigkeit
  - Synchronisierung oft schwierig

Verwendung

- Anwendungen in denen die Berechnungen Priorität haben
- Rechenintensive Berechnungen auf Multi-Core Systemen
- Programmteile können leicht in Java-Projekte eingebunden werden

Clojure vs. Java vs. OCaml

	Clojure	Java	OCaml
Objektorientiert	-	Ja	Ja, teilweise
Funktional	Ja	-	Ja
Nebenläufigkeit	Ja	Ja	-
Laufzeitumgebung	JVM	JVM	Eigene
Verfügbare Bibliotheken	Viele	Viele	Wenige

Features und kurze Einführung

- Dynamische Entwicklung
- Funktionale Features
- Lisp Features
- Polymorphismus
- Refsystem
- Nebenläufigkeit
- JVM

Dynamische Entwicklung

- Starten: java -cp clojure.jar clojure.main
- Clojure bietet ein Konsolen Interface
- Kein Kompilieren notwendig
- Codestruktur:

```
(def x 2)
#'user/x
(def y 20)
#'user/y
(+ x  y)
22
```

Funktionale Aspekte

- Unveränderbare Datentypen:
  - Listen
  - Vektoren
  - Sets
  - Maps
- Funktionen ohne Seiteneffekte
- Higher Order Funktion:

```
(def multiply (fn [x] (* 5 x)))
#'user/multiply
 (map multiply [1 2 3 4 5])
 (5 10 15 20 25)
```

Lisp Features & Polymorphismus

- Lisp Makrofeature erweitert
  - Makro erzeugen: „defmacro“
  - Makro erweitern: „macroexpand“
- Makros für „for“ und „while“ im Core integriert
- Polymorphismus wird unterstützt:
  - Unterstützung durch Multimethods: „defmulti“

Refsystem

- Um veränderbare Zustände zu realisieren
- Durch dereferenzieren einer Referenz erhält man dessen Inhalt
- Inhalt einer Referenz ist immer konsistent
- 4 Arten von Referenzen:
  - Agent
  - Atom
  - Ref
  - Var

Refsystem

	Synchron	Asynchron
Abhängig	Ref	
Unabhängig	Atom	Agent

- Synchron: Änderungen werden sofort vorgenommen
- Asynchron: Änderungen werden irgendwann in der Zukunft vorgenommen
- Abhängig: Vorgenommene Änderungen sind für andere Threads von Bedeutung
- Unabhängig: Änderungen für keinen anderen Thread von Bedeutung

Nebenläufigkeit

- Funktionen in Java Klassen kompiliert
- Interface Runnable und Callable implementiert
- Durch Java Threads ausführbar
- Ref‘s nur in Transaktionen veränderbar („dosync“)
- „Dosync“ verwendet ein System ähnlich zu ACID
- Immer konsistenter Zustand
- Kein Synchronisieren oder Sperren notwendig
- Funktion in Java Thread ausführen:
  - (.start(Thread. „Funktionsname“))

Nebenläufigkeit - Beispiel

```
(def account1 (ref 0))

(def account2 (ref 0))

(def transfer (fn [x]
        (dosync (let [a1 @account1 a2 @account2]
                (let [a1n (+ a1 x) a2n (- a2 x)]
                        (ref-set account1 a1n) (ref-set account2
                        a2n) "ok"
                )
        ))
))
```

JVM

- Alle Bibliotheken können genutzt werden
- Kein Speichermanagement
- Kein Verteilen der Aufgaben bei Multi-Core Systemen
- Beispiel:

  (javax.swing.JOptionPane/showMessageDialog nil "Hallo Welt!")

Community - Meinungen

- Relativ große Community

Positiv +	Negativ -
Lisp	Fehlermeldungen
JVM	Dokumentation
Nebenläufigkeit	

99 bottles of beer

```
(defn bottles-str [n]
        (str
                (cond
                        (= 0 n) "no more bottles"
                        (= 1 n) "1 bottle"
                        :else (format "%d bottles" n))
                 " of beer"))

(defn print-bottle [n]
    (println (format "%s on the wall, %s." (bottles-str n) (bottles-str n)))
    (println "Take one down and pass it around," (bottles-str (dec n)) "on the wall."))

(defn sing [n]
    (dorun (map print-bottle (reverse (range 1 (inc n)))))
    (println "No more bottles of beer on the wall, no more bottles of beer.")
    (println "Go to the store and buy some more," (bottles-str n) "on the wall."))

 (sing 99)
```

Zusammenfassung

- Lisp Dialekt
- Beruht auf der JVM
- Optimiert für Nebenläufigkeit
- Kein Sperren oder Synchronisieren notwendig