Seminar Report

Clojure

Georg Schmidhammer
georg.schmidhammer@student.uibk.ac.at

24 January 2013

Supervisor: Dr. René Thiemann

Abstract

Clojure is a very young functional LISP dialect which targets the JVM. Clojure tries to combine the power of LISP with the benefits of the JVM platform with all their libraries. Clojure is developed by Rich Hickey and was released 2007. In this article we will look at the benefits and deficits of this young programming language. Also we will compare Clojure to the two popular programming languages Java and Ocaml. To see what Clojure looks like there is also an introduction with small code examples.

1 Einleitung

Clojure ist eine Programmiersprache, die zu der Familie der LISP Dialekte gehört. Während LISP zu einer der ältesten Programmiersprachen zählt, ist Clojure noch eine sehr junge Sprache (2007 veröffentlicht). Clojure zählt zu den funktionalen Programmiersprachen, und läuft auf der JavaVirtualMachine (JVM).

Die Familie der Lisp Sprachen ist eng mit dem Lambda-Kalkül verbunden. Zu den bekanntesten Lisp Sprachen zählen Common Lisp und Scheme. Lisp wurde erstmals 1958 am MIT(Massachusetts Institute of Technology) spezifiziert.

Unter rein funktionalen Programmiersprachen versteht man Sprachen bei denen Programme aus Funktionen bestehen. Dabei ist wichtig, dass Variablen unveränderbar sind und der Rückgabewert von Funktionen ausschließlich von deren Eingabewerten abhängig ist. Des Weiteren sollten Funktionen keine Nebeneffekte (side effects) aufweisen. Solche unveränderlichen Variablen bieten den Vorteil, dass bei Multithreading unterschiedliche Threads sich nicht denselben Speicherbereich teilen. Ein weiteres Merkmal funktionaler Sprachen ist, dass der Code wie Daten behandelt wird.

Clojure zählt zwar zu den funktionalen Programmiersprachen, jedoch ist Clojure keine rein funktionale Sprache. Dies bedeutet, dass es in Clojure veränderbare Objekte und auch Funktionen mit Nebeneffekten gibt.

Mit Clojure wollte der Entwickler Rich Hickey einen Lisp Dialekt entwerfen, der auf einer bereits vorhandenen Plattform (die in Softwareprojekten auch häufig verwendet wird) aufbaut und den Schwerpunkt auf Nebenläufigkeit setzt. Somit fiel die Wahl recht schnell auf die JVM, da diese nach Ansicht des Entwicklers viele Vorteile wie Plattformunabhängigkeit, Speichermanagement und eine große Anzahl an Bibliotheken bereits zur Verfügung stellt.

2 Geschichte

Clojure wurde erstmals 2007 von Rich Hickey veröffentlicht. Bevor er mit der Arbeit an Clojure begann, arbeitete er an dem Projekt namens „dotLisp“, welches sehr ähnlich zu Clojure ist, jedoch auf dem .NET Framework basiert. Rich verbrachte etwa zweieinhalb Jahre mit der Arbeit an Clojure in denen er sich fast ausschließlich Clojure widmete. Im Jahr 2007, als der Großteil seiner Arbeit abgeschlossen war, sendete Rich Hickey ein Email an Freunde in der Common Lisp Community, in der er Clojure ankündigte.

Clojure gehört zu der Lisp Familie. Lisp ist nach Fortran die zweitälteste Programmiersprache die noch verwendet wird. Der Name Lisp steht dabei für List Processing (Listen Verarbeitung). Listen sind neben Atomen die Hauptbestandteile von Lisp Programmen. Diese könne beliebig verschachtelt werden. Und selbst Programmanweisungen sind Listen. Es gibt somit keinen Unterschied zwischen Daten und Programmanweisungen. Dadurch ist Lisp sehr flexibel und es können nahezu alle Datenstrukturen erstellt werden.

Rich Hickey ist allerdings kein großer Fan von objektorientierten Sprachen. Seiner Meinung nach sind veränderbare Objekte der Spaghetti Code der heutigen Zeit [2], da diese schnell unübersichtlich werden können und nur schwer zu testen sind.

3 Idee und Ziele

Rich Hickey entwickelte Clojure laut eigener Aussage, da er eine moderne Lisp Sprache wollte, die auf einer modernen Plattform basiert und für Nebenläufigkeit optimiert ist. Wie Rich sagte, fand er keine solche Sprache, worauf er eine eigene entwickelte [2].

Das Ziel von Clojure war somit klar gesteckt. Im nachfolgenden werden kurz die einzelnen Aspekte von Clojure geschildert und beschrieben, warum Rich Hickey sich dafür entschieden hat.

3.1 Lisp

Lisp Sprachen besitzen im Allgemeinen nur eine sehr klein Syntax und sind dadurch sehr schnell erlernbar.

Die bekanntesten Lisp Sprachen sind Common Lisp und Scheme. Diese Sprachen bauen jedoch nicht auf eine bereits vorhandene Plattform wie der (JVM) auf. Dadurch ist es schwieriger, in einem großen Softwareprojekt das z.B. in Java geschrieben ist, Programmteile in diesen Sprachen einzubinden.

3.2 Funktional

Unveränderbare Variablen, wie sie in funktionalen Sprachen üblich sind, erleichtern die Umsetzung von Nebenläufigkeit. Es ist kein Sperren oder Synchronisieren nötig, da nie auf denselben Speicherbereich zugegriffen wird [7]. Während in traditionellen Lisp Dialekten nur die Listen Strukturen rekursiv aufgebaut sind, sind bei Clojure alle Datenstrukturen rekursiv.

3.3 JVM

Die Vorteile der JVM sind, dass man sich die bereits vorhandenen Bibliotheken zu Nutze machen kann. Ebenso braucht man sich nicht um Speichermanagement kümmern, oder darum, wie die einzelnen Aufgaben bei Multi-CPU Systemen auf die Prozessoren verteilt werden.

3.4 Nebenläufigkeit

Bei traditionellen Java Programmen wird es dem Programmierer oft schwer gemacht, sein Programm wirklich Thread-safe zu machen. Auch wenn es mittlerweile zahlreiche Synchronisierungsverfahren und Thread sichere Datenstrukturen in Java gibt, muss immer noch der Programmierer selbst entscheiden, was er verwenden möchte. Eine falsche Entscheidung des Programmierers kann zu Code führen, der nicht Thread-safe ist. In Clojure braucht der Programmierer diese Entscheidungen nicht selbst treffen. Das Sperren und Freigeben von veränderbaren Objekten übernimmt Clojure selbst. Sollte es in Clojure dazu kommen, dass 2 Funktionen versuchen auf dasselbe veränderbare Objekt zu referenzieren, wird dieser Konflikt selbstständig erkannt und die 2 Funktionen sequentiell abgearbeitet [7].

4 Verwendung

Clojure wird vor allem bei Anwendungen verwendet, bei denen die Ergebnisse der Berechnungen absolute Priorität haben. Wie z.B bei Simulationen. Seine Stärken kann Clojure bei aufwändigen und rechenintensiven Anwendungen auf Multi-Core Systemen ausspielen [6].

Anwendungen und Berechnungen welche in kleinere Unterprobleme zerlegt werden können sind das bevorzugte Einsatzbereich von Clojure. Solange diese Teilprobleme unabhängig voneinander sind sollte der Speedup bei Clojure linear mit der Anzahl der verwendeten Kerne verlaufen.

In Bezug auf die Verwendung, ist ein großer Vorteil von Clojure, dass es auf der JVM beruht. Da eine große Anzahl von Softwareprojekten in Java geschrieben sind, ist es dort leicht möglich Programmteile die in Clojure geschrieben sind einzubinden.

5 Clojure vs. Java vs. OCaml

Während Clojure und Java auf der JVM aufsetzen, setzt OCaml auf einer eigenen Plattform auf. Java ist eine Objekt-orientierte Sprache, Clojure und OCaml sind hingegen funktionale Sprachen.

Die JVM übersetzt den Programmcode von Java und Clojure in Bytecode. Dies macht den Code auf allen Systemen ausführbar auf denen Java installiert ist. Und da Java für alle gängigen Betriebssysteme verfügbar ist, ist Clojure und Java plattformunabhängig. OCaml Code hingegen kann in Bytecode und Maschinencode übersetzt werden. Durch die Übersetzung in Maschinencode ist es möglich, den Code direkt auf dem verwendeten System auszuführen ohne dass, eine virtuelle Maschine auf dem System installiert ist. Da das Übersetzen in Maschinencode jedoch dazu führen würde, dass der Code von OCaml nicht plattformunabhängig ist, ist es auch möglich, OCaml Programme in Bytecode zu Übersetzen und mittels der eigenen Laufzeitumgebung auszuführen.

Alle 3 Sprachen verwenden Bibliotheken um zusätzliche Funktionen zu ermöglichen. Bei der Anzahl der verfügbaren Bibliotheken, gibt es allerdings Unterschiede. Für die JVM von Java gibt es unzählige Bibliotheken. Die Bibliotheken reichen dabei von komplizierten mathematischen Operationen, bis zu Bibliotheken für grafische User Interfaces wie Swing oder AWT [5]. Da Clojure auf der JVM basiert, können in Clojure Programmen alle Java Bibliotheken verwendet werden. Auch wenn es für OCaml ebenfalls viele Bibliotheken gibt, sind es doch deutlich weniger als für Java. Viele Bibliotheken erweitern dabei lediglich Funktionen die bereits in der umfangreichen Standardbibliothek von OCaml enthalten sind. Eine Auswahl an OCaml Bibliotheken, kann im Internet gefunden werden [4].

Java und Clojure machen sich im Bereich Nebenläufigkeit die JVM zu Nutze. Diese kümmert sich darum, wie die Threads auf die Kerne aufgeteilt werden und um die gesamte Speicherverwaltung der Threads. OCaml hingegen unterstützt keine Form von Nebenläufigkeit. Java ist eine der meist verwendeten objektorientierten Programmiersprachen. Auch Ocaml ist zum Teil objektorientiert. Clojure hingegen unterstützt keinerlei Objektorientierung. In der unten angeführten Grafik werden die Unterschiede und Gemeinsamkeiten der 3 Programmiersprachen Java, Clojure und OCaml nochmals übersichtlich dargestellt.

	Clojure	Java	OCaml
Objektorientiert	-	Ja	Ja, teilweise
Funktional	Ja	-	Ja
Nebenläufigkeit	Ja	Ja	-
Verfügbare Bibliotheken	Viel	Viel	Wenige

6 Features und kurze Einführung

Clojure bietet zahlreiche Features welche zum Großteil von der recht eng verwandten Sprachfamilie Lisp stammen. Jedoch macht sich Clojure auch einige Fähigkeiten der JVM zu nutzen.

6.1 Dynamische Entwicklung

Wie bereits beschreiben, können Clojure Programme in Java oder JavaScript Programmen eingebettet werden. Jedoch bietet Clojure auch ein Konsolen Interface, welches auf den Namen Read-Eval-Print-Loop (REPL) hört. Dieses Interface erlaubt es dem Benutzer, Kommandos eizugeben und auszuführen. Die Übersetzung erfolgt dabei zur Laufzeit. Die REPL kann mittels folgenden Befehls gestartet werden:

java -cp clojure.jar clojure.main

Nach dem Start erhält man als Ausgabe „user=>“.
Die Grundlegende Struktur von Clojure Kommandos sieht wie folgt aus:

Beispiel1:

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

Durch def werden in Clojure Namen zugewiesen. Mittels #‘user/ signalisiert Clojure dem Benutzer dass eine Funktion oder Variable nun angelegt wurde. Nach dem Schrägstrich, steht dabei immer der Name der angelegten Funktion oder Variable. Durch („Name“) wird eine Funktion ausgeführt, oder wie im Beispiel 1, eine bereits vorhanden Operation (+) auf 2 Werte angewendet. Kommentare können in Clojure durch einen Strichpunkt am Anfang einer Zeile geschrieben werden.

6.2 Funktionale Aspekte

Da Clojure eine funktionale Programmiersprache ist, bietet Clojure natürlich Möglichkeiten an, unveränderbare Objekte zu erzeugen. Ebenso können Funktionen ohne jegliche Seiteneffekte erzeugt werden.

Mit „fn“ werden Funktions Objekte erzeugt. In den eckigen Klammern [] können dabei Parameter übergeben werden. Da bei Clojure Code wie Daten behandelt werden, können diese natürlich auch wiederum an Funktionen übergeben werden, oder in eine Variable gespeichert werden. In Beispiel 2 wird eine Funk-

tion multiply definiert, die einen Parameter erhält und diesen mit 5 multipliziert. Map ist eine in Clojure bereits integrierte High-Order Funktion. Die map Funktion iteriert über eine Liste von Zahlen und wendet darauf die Funktion multiply an. Anschließend gibt die map Funktion eine Liste mit Resultaten zurück.

Beispiel2:

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

Mittels „let“ können lokale Werte gespeichert werden. Die mittels „let“ erzeugten Werte sind jedoch keine Variablen. Denn, wenn ihnen einmal ein Wert zugewiesen wurde, ändert sich dieser Wert nie wieder.

Um die Unveränderbarkeit von Objekten zu gewährleisten, bietet Clojure zahlreiche unveränderbare Datenstrukturen wie Listen, Vektoren, Sets und Maps an. Wenn diesen Strukturen etwas hinzugefügt oder etwas daraus gelöscht wird, verändert sich das Objekt nicht. Die Struktur wird dabei mit den zusätzlichen oder fehlenden Werten neu erstellt. Dabei wird die alte Struktur nicht kopiert, da dies zu zeitintensiv wäre. Deshalb sind diese Strukturen mittels verlinkten Listen implementiert. Dies ermöglicht ein schnelles neu erstellen der Struktur. Bei diesen Veränderungen der Strukturen bleiben die alten Strukturen jedoch bestehen.

In Beispiel 3 wird ein Vektor(my-vector), eine Map(my-map) und eine Liste(my-list) erstellt und darauf Operationen ausgeführt. Die Funktion „conj“ nimmt ein Element entgegen, und fügt dies einer Liste oder einen Vektor hinzu. Zurückgegeben wird dabei ein neuer Vektor oder Liste. Die Funktion „assoc“ nimmt einen Schlüssel und einen Wert entgegen und fügt diese, einer map hinzu. Zurückgegeben wird eine neue map. Der key muss dabei mittels „:“ gekennzeichnet werden und der Wert muss unter Anführungszeichen stehen. Während bei Listen und Vektoren, die Werte in eckige Klammern geschrieben werden, werden die Werte bei Maps in geschlungene Klammern geschrieben. Wie man im Beispiel 3 sehen kann, bleiben bei diesen Operationen die alten Listen, Vektoren und Maps bestehen.

Beispiel3:

```
(let [my-vector [1 2 3 4]
        my-map {:fred “ethel”}
```

```
        my-list (list 4 3 2 1)]
    (list
        (conj my-vector 5)
        (assoc my-map :ricky "lucy")
        (conj my-list 5)
        ;the originals are intact
        my-vector
        my-map
    my-list))
([1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (5 4 3 2 1) [1 2 3 4] {:fred "ethel"} (4 3 2 1))
```

6.3 Lisp Features

Als Teil der Lisp Familie, setzt Clojure auch zahlreiche aus Lisp bekannte Funktionalitäten um und erweitert diese sogar. So wird das Lisp Makro System verwendet und die Eigenschaft, dass Code wie Daten behandelt wird, wurde noch auf Vektoren und Maps erweitert. Makros werden verwendet um Wiederholungen im Code zu vermeiden. Makros können mittels des Befehls „defmacro“ [1] oder „macroexpand“ [1] erzeugt werden. Makros können auch verwendet werden, um und-Verknüpfungen oder den for-Operator zu definieren. Auch wenn Clojure von sich aus keine for und while Schleifen kennt, sind zahlreiche Operatoren im Core bereits als Makros definiert. Diese können von den Benutzern verwendet werden ohne sie selbst implementieren zu müssen. So sind die Makros „for“ und „while“ bereits seit Version 1.0 im Core integriert. All diese Makros sind in der Clojure API genau definiert [1].

6.4 Polymorphismus

Clojure unterstützt Polymorphismus durch sogenannte „Multimethods“. Diese können mittels des Befehls defmulti erzeugt werden [1]. Mit diesen Multimethods können Funktionen überlagert werden. Somit ist die Ausgabe der Funktionen abhängig von dem Typen der Parameter die, die Funktion erhält. Hier ein Beispiel, wobei die Funktion encounter überlagert wird. Abhängig davon, welche Tiere der Funktion übergeben werden, werden unterschiedliche Meldungen ausgegeben.

6.5 Refsystem

Der funktionale Ansatz von Clojure schließt eigentlich veränderbare Datenstrukturen aus. Da diese jedoch trotzdem oft benötigt werden, wurde ein Refsystem integriert. Dieses Refsystem funktioniert, indem ein Symbol auf eine unveränderliche Ref-Var gebunden wird. Diese Ref-Var besitzt eine veränderbare Referenz auf eine andere Var. Durch derefenzieren mittels des Befehls „deref“ kann auf den Inhalt der Variable zugegriffen werden. Da der Inhalt einer solchen Var immer konsistent ist, braucht der Zugriff auf eine solche Var nicht synchronisiert werden. Eine Veränderung einer solchen Ref kann nur im Rahmen einer Transaktion erfolgen. Ansonsten sind Refs unveränderbar und an eine bestimmte Speicherstelle geknüpft.

In Clojure gibt es 4 verschiedenen Arten solcher Referenzen [6]:

- Agent: Agents werden für asynchrone, unabhängige Änderungen an veränderbaren Objekten verwendet. Unabhängig bedeutet, dass die Änderungen die vorgenommen werden, für keinen anderen Thread von Bedeutung sind. Asynchron bedeutet, dass die Änderungen nicht sofort umgesetzt werden, sondern irgendwann in der Zukunft.
- Atom: Atoms werden für synchrone, unabhängige Änderungen an veränderbaren Objekten verwendet. Synchron bedeutet dabei, dass die Änderungen sofort vorgenommen werden.
- Ref: Atoms werden für synchrone, abhängige Änderungen an veränderbaren Objekten verwendet. Abhängige Änderungen, bedeuten, dass die Änderungen für andere Threads wichtig sind. Ein Beispiel, dafür wäre z.B. die Überweisung von Geld von einem Konto auf ein anderes.
- Var: Vars werden im Hintergrund automatisch verwendet ohne, dass man dabei etwas bemerkt. So wird beispielsweise bei (def n 1), n automatisch als Name für den eine Var definiert, die den Wert 1 besitzt. Durch (def n 2), wird nicht die Referenzierung von n zu Var verändert, sondern lediglich der Wert der Var auf 2 gesetzt.

In Beispiel 4 wird das erzeugen eines Atoms gezeigt und die Änderung des Wertes mittels „reset!“. Durch den Befehl „@“ wird der println-Funktion mitgeteilt, dass der Wert des Atoms ausgegeben werden soll.

Beispiel4:

```
(def my-atom (atom 1))
(reset! my-atom 2)
2
(println @my-atom)
2
```

Übersicht der Referenztypen:

	Synchron	**Asynchron**
Abhängig	Ref	
Unabhängig	Atom	Agent

6.6 Nebenläufigkeit

Da die in Clojure enthaltenen Core Datenstrukturen alle unveränderbar sind, können sie von unterschiedlichen Threads verwendet werden. Dabei ist kein Sperren der Datenstrukturen nötig. Clojure verwendet die Threads der JVM. Dadurch muss sich der Programmierer nicht darum kümmern wie die Threads auf die Prozessorkerne aufgeteilt werden. All dies übernimmt die JVM. Alle Funktionen in Clojure werden dabei in Java Klassen kompiliert welche die Inferfaces Runnable und Callable implementieren. Dadurch sind Clojure Funktionen parallel ausführbar. Beispiel 5 zeigt wie man eine erzeugte Funktion mittels eines Java Threads ausführen kann. Die verwendete Funktion „println“ führt dabei zu einer Ausgabe auf der Konsole. Mit „Thread.“ wird eine Thread erzeugt und die erzeugte Funktion „hello“ übergeben. Mit „.start“ wird der Thread gestartet.

Beispiel5:

```
(defn hello [] (println ”Hallo”))
# 'user/hello
user= >(.start (Thread. hello))
nil
Hallo
```

Die in Clojure verwendeten Variablen(z.B. (def x 2)) können in nebenläufigen Programmen verwendet werden, da diese Variablen Thread Isolation verwenden. Das bedeutet, dass alle anderen Threads die Veränderungen die Thread 1 an der Variable vorgenommen hat erst sehen, wenn Thread 1 die Änderungen an der Variable abgeschlossen hat.

Die in Clojure enthaltenen Transaktions-Referenzen (Refs) verwenden das software transactional memory System(STM), um Thread Sicherheit zu gewährleisten. Das STM sorgt dafür, dass das Ändern von Zuständen immer zu einem konsistenten Zustand führt. Dabei verwendet das STM das aus Datenbanken bekannte ACID Prinzip. Dadurch wird sichergestellt, dass alle Aktionen, die auf Refs ausgeführt werden atomar, konsistent und isoliert sind. Sollte es dazu kommen, dass 2 Threads gleichzeitig eine Referenz verwenden wollen, muss ein Thread eine bestimmte Zeit warten und versucht es anschließend erneut. All dies geschieht ohne explizite Locks von Seiten des Benutzers.

In Beispiel 5 wird ein Ref erzeugt und mittels „ref-set“ verändert. Der Befehl „ref-set“ muss dabei innerhalb eines „dosync“ stehen. Das „dosync“ ist dabei die Transaktion. „Nil“ steht für den Anfangswert der Erzeugten Ref. Nil ist dasselbe wie „null“ in einem Java Programm [1].

Beispiel5:

```
(def ref1 (ref nil))
# 'user/ref1
(dosync (ref-set ref1 5))
@ref1
5
```

In Beispiel 6 werden Ref's verwendet um das Überweisen von Geld zu simulieren. Am Anfang werden 2 Accounts erstellt, mit einem ref mit Startwert „0“. Die Funktion transfer bekommt einen Parameter x (Summe die überwiesen wird). Die Überweisung erfolgt dabei immer von account2 nach account1. Um die Ref's zu verändern bedarf es einer „dosync“-Transaktion. Diese Transkation sorgt dafür, dass die Überweisung immer zu einem konsistenten Zustand führt und kein anderer Thread gleichzeitig die ref's der Accounts verändert. In diesem Beispiel werden in der main Funktion 1000 Threads erzeugt, welche eine Transaktion ausführen. In einer Transaktion werden 10 Euro überwiesen. Nach diesen Transaktionen werden die Kontostände der Accounts mittels „(list @account1 @account2)“ ausgegeben.
Beispiel6:

```
(def account1 (ref 0))
# 'user/account1
(def account2 (ref 0))
# 'user/account2
(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"
                    )
            ))
))
# 'user/transfer

(defn main []
            (dotimes [n 1000] (.start(Thread. (transfer 10))))
            (list @account1 @account2)
)
```

Ausgabe des Programms:
(10000 -10000)

In einem herkömmlichen Java Programm wäre es nötig die Accounts und die Überweisung zu synchronisieren. Ansonsten würden mit größter Wahrschein-

lichkeit, 2 Threads gleichzeitig eine Transaktion ausführen. Dies würde dazu führen, dass eine oder mehrere Transaktionen verloren gehen würden. In Java würde es mit größter Wahrscheinlichkeit dazu kommen, dass 2 Threads den gleichen Kontostand lesen, und diesen inkrementieren. Anschließend würden die Threads nacheinander, den von ihnen berechneten neuen Kontostand schreiben. Somit wäre die Überweisung des Threads, der als erster den neuen Kontostand schreibt verloren. In Clojure kann diese Szenario nicht eintreten. Auch ist in Clojure keine Synchronisierung notwendig. Die „dosync“ Transaktion verhindert, dass Überweisungen verloren gehen. Alle entstehenden Konflikte werden von Clojure selbst aufgelöst.
Ausgabe des Beispiels:
Startwert
Endwert

Beispiel 7 zeigt ein Programm mit Agents, welches einen Counter simuliert. Dabei wird ein Agent (counter), mit Startwert 10000 erzeugt. Anschließend wird eine Funktion count definiert. Diese Funktion sendet eine Funktion mittels der Funktion „send“ an den Agent. Die Funktion die an den Agent gesendet wird, wartet zuerst den Wert von Agent in Millisekunden, bevor sie diesen Wert um 1 erhöht. Der Wert wird mittels der vordefinierten Funktion „inc“ erhöht. In der Main-Funktion wird zuerst die Funktion count aufgerufen. Anschließend wird der Wert des Agents ausgegeben. Dann wird 11 Sekunden lang gewartet, bevor der Wert erneut ausgegeben wird.

Beispiel7:

```
(def counter (agent 10000))
# 'user/counter
(defn count [] (send counter #(do (Thread/sleep % 1) (inc % 1))))
# 'user/count
(defn main []
      (count)
      (@counter)
      (. java.lang.Thread sleep 11000)
      (@counter)
)
```

Ausgabe des Programms:
#<Agent@17e21e3: 10000>
(10000)
(10001)

In diesem Beispiel kann man sehen, dass der Wert des Agents erst nach Ablauf

der Zeit verändert wird. Vor Ablauf dieser Zeit, liefert der Agent den konsistenten alten Wert zurück. Statt der sleep Funktion könnte man auch den Befehl „(await counter)“ verwenden. Dieser Befehl wartet, bis der Counter inkrementiert ist und fährt erst dann fort.

Nach diesen Beispielen mit Refs und Agents stellt sich nun die Frage wann wir diese verwenden. Refs werden immer dann verwendet, wenn man einen Zustand oder Wert benötigt, der von mehreren Threads verändert wird. Agents hingegen werden für Thread interne Zustände wie beispielsweise einen Counter verwendet. Dieser Counter ist für keinen anderen Thread relevant. Dadurch, dass Agents asynchron sind, hat man auch keine Garantie darüber, wann der Wert effektiv geändert wird.

6.7 JVM

Da Clojure auf der JVM basiert, kann sich der Programmierer alle von Java bekannten Bibliotheken zu nutzen machen. Die von der JVM zur Verfügung gestellten Threads macht sich Clojure auch zu Nutzen. Alle in Clojure geschriebenen Funktionen werden zu Java Bytecode kompiliert. Auch die Swing Bibliothek von Java kann man verwenden wenn man eine grafische Oberfläche gestalten möchte.

So führt der Code:
(javax.swing.JOptionPane/showMessageDialog nil ”Hallo Welt!”)
Zu folgender Ausgabe am Bildschirm:

Figure 1: Hallo Welt

7 Community Meinungen

Auch wenn Clojure noch recht jung ist, besitzt es bereits eine recht ansehnliche Community. Durch diese Community bekommt man auch einen recht guten Einblick, was Programmierer gut und schlecht finden.

Die LISP Eigenschaften von Clojure stoßen in der Community auf große Zus-

timmung. Dies liegt allerdings auch daran, dass ein Großteil der Community und auch der Entwickler Rich Hickey selbst, aus dem LISP Hintergrund kommen. Clojure wurde auch erstmals im Common Lisp Forum angekündigt und veröffentlicht. Die Mächtigkeit dieser Sprachfamilie wird von den meisten Usern sehr geschätzt. Für Neulinge auf diesem Gebiet ist Clojure jedoch relativ schwer erlernbar. Auf jemanden der noch nie mit funktionalen Programmiersprachen oder einer Sprache aus der LISP Familie gearbeitet hat, wirkt der Code anfangs wenig strukturiert und verwirrend.

Die Verfügbarkeit der vielen Bibliotheken ist laut der Community ein gutes Feature. Jedoch werden viele verfügbare Bibliotheken so gut wie nie verwendet. So ist es z.B. möglich grafische Anwendungen mittels der Swing Bibliothek zu schreiben. Jedoch ist solch ein Programm in Clojure schwerer zu erstellen als in Java und bringt auch keine Vorteile gegenüber der Java-Variante. Jedoch ist dies nicht als Manko an Clojure zu sehen, da es schlussendlich an dem Benutzer liegt, welche Bibliotheken er verwenden möchte.

Ein ernsthaftes Problem sind jedoch die noch nicht sehr ausgereiften Fehlermeldungen. So kommt es laut Clojure Programmierern immer wieder vor, dass in einer Fehlermeldung eine nicht existente Zeile angegeben wird, oder generell Fehlermeldungen ausgegeben werden, die nicht Ursache des Problems sind. Dies ist für Programmierer ein großes Problem, da es dadurch sehr zeitaufwendig ist, den Fehler ausfindig zu machen.

Ein weiteres in der Community bemängeltes Manko ist die Dokumentation. Auch wenn die Dokumentation für eine solch junge Sprache sehr umfangreich ist, kann sie nicht mit einer Java Dokumentation mithalten. So fehlt beispielweise eine gute Suchfunktion und eine übersichtlichere Struktur der Dokumentation wäre auch wünschenswert. Neben der Dokumentation auf der Clojure Homepage, gibt es eine Community Dokumentation namens ClojureDocs [3]. Diese befindet sich jedoch noch im Beta Stadium. Der Ansatz dieser Dokumentation ist gut, jedoch fehlt es an Usern. Dies erkennt man daran, dass man zwar fast alles findet, jedoch häufig Beispiele und genauere Beschreibungen fehlen.

Das am meisten genutzte Feature von Clojure ist dessen Nebenläufigkeit. Darin sehen die meisten User den größten Nutzen. Ein Grund dafür ist, dass es nicht nötig ist, Programmteile zu synchronisieren oder zu sperren. In Java können falsch angewandte Synchronisierungsmechanismen zu Deadlocks, sehr langsamen Code oder zu falschen Ergebnissen führen. All dies ist in Clojure nicht möglich und wird deshalb so geschätzt.

8 Entwicklungsgrund

Die Entwicklung von Clojure war nicht wirklich nötig um ein gewisses Ziel zu erreichen. Viel mehr war es die Einstellung von Entwickler Rich Hickey, die ihn zwang, Clojure zu entwickeln. Wie er selbst behauptet, war er genervt

vom jahrelangen Programmieren in Java und C#. Ebenso ist er kein Fan von veränderbaren Variablen. Da er keinerlei Sprache finden konnte, die genau seinen Vorstellungen entsprach, begann er seine eigene Programmiersprache zu entwickeln.

9 Alternativen

Alternativen, die genau dieselben Features bieten wie Clojure gibt es keine. Dies ist auch logisch, da Rich Hickey Clojure entwickelte, da es keine solche Sprache gab. Wenn man jedoch einige kleine Kompromisse eingeht findet man eine Vielzahl an Alternativen.

Da wären zum einen zahlreich LISP Dialekte, für Programmierer die vor allem dieses Feature von Clojure schätzen. Die wohl bekannteste Alternative ist Common Lisp. Common Lisp ist eine funktionale Sprache wie Clojure. Die meisten LISP Dialekte können jedoch mit Clojure in Sachen Nebenläufigkeit nicht mithalten.

Häufig wird Clojure auch mit Scala verglichen. Scala ist eine funktionale, objektorientierte Programmiersprache. Jedoch liegt der Schwerpunkt von Scala nicht auf Nebenläufigkeit. Zudem schätzen viele Anhänger von Clojure die Tatsache, dass es bewusst nicht objektorientiert ist. .

10 Ziel

Die Ziele von Clojure hat sich lediglich der Entwickler selbst gesteckt. Und sein Ziel war es, einen funktionalen LISP Dialekt zu erstellen, der auf einer ausgereiften Plattform beruht. Des Weiteren sollte die Sprache auf Nebenläufigkeit optimiert werden. Dieses Ziel hat Rich Hickey zum Großteil erreicht. An der relativ großen Community erkennt man, dass viele Personen Clojure für gut befinden. Clojure wird auch immer noch weiterentwickelt. Zurzeit ist die Version 1.5 in Arbeit. Clojure soll noch einfacher und schneller werden.

11 Zusammenfassung

Die Idee hinter Clojure, einen modernen funktionalen LISP Dialekt zu erschaffen, der auf der JVM beruht, ist äußerst interessant. Mit dem Schwerpunkt auf Nebenläufigkeit ist Clojure auch zeitgemäß. Prozessoren werden mit immer mehr Kernen ausgestattet. Um die Leistung der neuesten Prozessoren auch vollkommen auszunutzen, bedarf es immer besserer Unterstützung von Nebenläufigkeit. Jedoch braucht sich der Entwickler in Clojure keine Gedanken über Deadlocks und Locking-Mechanismen zu machen, wenn er sein Programm parallelisieren möchte.

Die Entscheidung bei Clojure, komplett auf Objektorientierung zu verzichten, ist allerdings fragwürdig. Denn Objektorientierung trägt viel zur Strukturierung von Programmen bei. Die meisten Programmierer, die große Software Projekte realisieren, wollen nicht mehr auf dieses Feature verzichten. Selbst Common Lisp bietet mit Common Lisp object system (CLOS) eine objektorientierte Erweiterung. Jedoch wurde in Clojure bewusst auf die Objektorientierung verzichtet, wodurch eine nachträgliche Erweiterung wohl ausgeschlossen werden kann.

Die Tatsache, dass in LISP Dialekten Code wie Daten behandelt werden, macht sie sehr mächtig, da jede Datenstruktur erstellt werden kann. Dies wird von vielen Programmierern sehr geschätzt. Deshalb kommt auch ein Großteil der Clojure Nutzer aus dem LISP Bereich. Jedoch bringt solch ein LISP Dialekt auch Nachteile. Denn für unerfahrenen Benutzer die lediglich Java und C beherrschen ist die Einstiegshürde groß und Clojure schwerer zu erlernen.

Für alle die bereits mit LISP Dialekten oder funktionalen Programmiersprachen vertraut sind, lohnt sich ein Blick auf Clojure allemal. Da es interessante Aspekte von LISP, funktionalen Sprachen und der JVM versucht zu vereinen.

References

[1] Clojure api. http://clojure.github.com/clojure/clojure.core-api.html, 2013.

[2] Clojure homepage. http://clojure.org, 2013.

[3] Clojuredocs. http://clojuredocs.org/, 2013.

[4] Ocaml libraries. http://caml.inria.fr/cgi-bin/hump.en.cgi?sort=0&browse=66, 2013.

[5] P. Gestwicki and B. Jayaraman. Interactive visualization of java programs. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 226 – 235, 2002.

[6] M. Kalin and D. Miller. Clojure for number crunching on multicore machines. *Computing in Science Engineering*, 14(6):12 –23, nov.-dec. 2012.

[7] Massimo Di Pierro and David Skinner. Concurrency in modern programming languages. *Computing in Science and Engineering*, 14(6):8–10, 2012.