Clojure

vorgelegt am:	12. Mai 2011
Studienbereich:	Informatik
Fakultät:	Vermessung, Informatik und Mathematik
Bildungseinrichtung:	Hochschule für Technik Stuttgart
von:	Benjamin Britsch, Alain M. Lafon
Matrikelnummer:	373679, 372991
Prüfer:	Prof. Dr. Stefan Knauth, HfT Stuttgart

Zusammenfassung

In Zeiten in denen davon gesprochen wird, dass Moore's Law in absehbarer Zukunft nicht mehr gilt werden zusehens Architekturen eingesetzt, die nicht mehr darauf vertrauen, dass jede einzelne CPU ausreichend schnell ist, sondern vielmehr massiv parallele Rechenleistung bieten. Im Zuge dieses Architekturwandels müssen gleichwertig die eingesetzten Software-Stacks überdacht und teilweise neu ausgelegt werden, um die neue Form von Rechenleistung konsolodiert und effizient zu nutzen.

Clojure ist ein LISP Dialekt und damit eine primär funktionale Programmiersprache, die es sich zum Ziel genommen hat dem Entwickler und der Anwendung hohe Parallelität und Performanz bei gleichzeitig hohem Komfort bereitzustellen. Diese Arbeit bietet einen Überblick über die dahinter stehende Methodik und grenzt sie gegen den bisherigen Verfahren ab

Inhaltsverzeichnis

Inhaltsverzeichnis

Abkürzungsverzeichnis

XML ..... Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation und Ziel der Arbeit

Die vorliegende Arbeit ist die schriftliche Ausarbeitung zum Thema **Clojure**. In ihr wird Clojure gegenüber LISP abgegrenzt und über die Vorteile dynamischer und funktionalier Programmierung im Gegenzug zu konvetiell strukturierter Programmierung besprochen.

1.2 Typographische Konventionen

Folgende typographische Konventionen sind in dieser Arbeit eingesetzt.

- **Neuer Begriff**
  Neue Begriffe sind für den schnellen Überblick gesondert im Textbild hervorgehoben.
- *Fachbegriff*
  Fachbegriffe sind aus dem selben Grund wie neue Begriffe hervorgehoben.
- `Eingabe`
  Referenzen auf Tastatureingaben sind als solche gekennzeichnet.
- `Quellcode`
  Quellcode kann wie eine Eingabe im Text eingebettet werden. Bei größeren Code-Versatzstücken wird jedoch volles Syntax-Highlighting verwendet.
- `C:\Pfad\Datei`
  Pfad- und Datei-Angaben

1 Einleitung

- Datentyp
  Referenzen auf interne Datenstrukturen und Variablennamen tragen die typographische Kennzeichnung Datentyp.

2 Funktionale Programmiersprache

2.1 Definition

Die heutigen funktionalen Programmierparadigmen beruhen auf dem in den 1930er Jahren von Alonso Church und Stephen Kleene eingeführten **Lambda-Kalkül**. Es handelt sich dabei um eine formale Sprache um Funktionen zu untersuchen. Basierend darauf wurde 1960 von Marvin Minsky LISP entwickelt, die sozusagen erste funktionale Programmiersprache.

Heutzutage ist es schwer, eine allgemeingültige Definition für Funktionale Programmiersprachen zu finden, da funktionale Konzepte in sehr vielen Programmiersprachen vorkommen. In der Wikipedia wird der Begriff wie folgt definiert:

"Eine funktionale Programmiersprache ist eine Programmiersprache, die Sprachelemente zur Kombination und Transformation von Funktionen anbietet. Eine rein funktionale Programmiersprache ist eine Programmiersprache, die die Verwendung von Elementen ausschließt, die im Widerspruch zum funktionalen Programmierparadigma stehen." wikipedia.org

Der erste Teil der Definition trifft aber auf sehr viele Programmiersprachen zu, die man im Allgemeinen nicht als funktional bezeichnen würde. Der zweite Teil hingegen schließt unter Umständen sehr viele Sprachen aus, die man wiederum im Allgemeinen als funktional bezeichnen würde, je nach dem was man alles zum funktionalen Programmierparadigma zählt.

Im Gegensatz dazu umfasstttt die Definition der School of Computer Science Nottingham mehr die Sprachen, die auch umgangssprachlich als funktional bezeichnet werden.

"Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style." of Computer Science Nottingham

Die Betonung liegt hier auf **encourage**, also das Anregen oder Ermutigen, auf funktionale Weise zu Programmieren. Damit werden all die Sprachen aus der Definition ausgeschlossen, die nur aus einem imperativen Ansatz heraus funktionale Elemente implementieren.

2.2 Vorteile

- **Andere Sichtweise auf Probleme:** Man versucht nicht, wie bei imperativer Programmierung, zu beschreiben, welche Operationen auf dem Speicher ausgeführt werden, sondern man versucht vielmehr den Lösungsweg selber anzugeben. Diese Sichtweise kann dann durchaus auch bei der Programmierung in imperativen Sprachen helfen, sofern diese funktionale Konzepte zumindest Teilweise unterstützen. Das ist aber bei sehr vielen weit verbreiteten Sprachen der Fall.
- **Eleganter:** Die meisten Problemlösungen sind deutlich kürzer und eleganter wenn man sie funktional implementiert, da kein Zustand betrachtet werden muss. Der Code ist deshalb auch übersichtlicher.
- **Debuggen und Testen einfacher:** In rein funktionalen Programmiersprachen müssen weder Seiteneffekte noch Zustände berücksichtigt werden. Beim Debuggen können Fehler deshalb unabhängig vom Speicherzustand betrachtet werden. Durch die hohe Modularität kann man auch sehr einfach einzelne Funktionen testen.
- **Einfache Syntax:** Im Allgemeinen haben funktionale Programmiersprachen eine einfachere Syntax als imperative Programmiersprachen, da sie mit wenigen syntaktischen Konstrukten und wenigen Schlüsselwörtern auskommen.

- **Modularisierung:** Ein Problem kann soweit zerlegt werden, dass es nur noch aus simplen Funktionen besteht, die dann wiederum in anderen Problemlösungen verwendet werden können.
- **Parallelisierbarkeit:** Einzelne Teilausdrücke können ohne Probleme parallel ausgewertet werden. Das ist insbesondere ein großer Vorteil bei der Programmierung für Multicore Systeme.

3 Lisp

3.1 Definition

LISP wurde 1960 von John McCarthy in einer wissenschaftlichen Arbeit forciert. Diese Arbeit zeigt auf wie durch einige wenige einfache Operatoren und einer definierten Notation von Funktionen eine vollwertige Programmiersprache definiert wird. Eine von McCarthy's grundlegend neuen Ideen war eine einfache Datenstruktur sowohl für Code als auch für Daten zu verwenden. Daher der Term *LISP* - er steht für "List processing".

McCarthy definierte eine Sprache, die auf nur sechs Grundfunktionen fußte - car, cdr, cons, cond, lambda und Rekursion. Dabei gab es damals weder verbreiteten Nutzen von lambda Berechnungen noch von Rekursion. Noch vor McCarthy´s Lisp wurde bei IBM debattiert zu welchem Zweck Rekursion eigentlich dienen möge und kam zu keinem sinnigen Schluss. McCarthy klärte diese Fragen jedoch schon in den ersten Vorträgen in denen er LISP vorstelltePaul Graham [a].

In der Essenz gibt heute zwei Paradigmen Programme zu schreiben - der eine Zweig bedient sich dem Vorbild C's und den aus diesem Gedankengut entstandenen Sprachen, der andere Zweig sind LISPs und LISP ähnliche Sprachen. *LISP* selbst wurde lange Zeit vernachlässigt - aus guten und weniger berechtigten Gründen. Doch je weiter die Zeit fortschreitet verwenden neue Sprachen zusehens Denkweisen und Modelle aus *LISP* - beispielsweise "Garbage Collection" und "Runtime Typing". Der Ansatz des *Code als Daten* und der Macros heben LISP allerdings bis heute eindeutig aus der restlichen Sprachvielfalt hervor.

3.2 Abgrenzung Lisp/Java/Clojure

Clojure baut auf dem Stack der *Java Virtual Machine* auf. Dies bietet einerseits den Entwicklern der Sprache selbst den Vorteil viele Probleme der Informatik nicht selbst lösen zu müssen, sondern auf solide Grundlagen wie Datentypen, Threading oder Portabilität der JVM zurückgreifen zu können. So können sie sich ganz der Sprache und interessante neue Probleme konzentrieren statt etwa einen eigenen Garbage Collector schreiben zu müssen. Anwender der Sprache hingegen kommt dieser Ansatz ebenfalls zu Gute - die Sprache schnell weiter und kann so in kurzen Iterationen auf neue Eingaben aus der Community reagieren. Abseits von diesem grundlegenden Vorteil kommt ein wichtiger Business Aspekt: Clojure kann direkt mit dem Java Ökosystem interagieren und hat so Zugang zu einem reichen Quell bestehender Software, Bibliotheken und Datenbanken.

Mit diesem Modell lößt Clojure gleich mehrere Probleme. Auf Seiten der Sprache selbst bietet sich eine grundsolide Platform, die zugleich performant wie portabel ist. Hierfür hat Sun Microsystems über mehr als ein Jahrzehnt hinweg mehr Resouren investiert als einigen wenigen Entwicklern einer neuen Sprache möglich ist. Wissend um diese Vorteile der JVM gibt es viele Programmierer, die bisher willentlich darauf verzichtet haben, da Java verglichen mit Alternativen weder unbedingt als schön, bündig oder leicht bezeichnet werden kann.

Viele dieser Programmier suchten bisher Zuflucht in anderen Sprachen wie Python oder Ruby, die noch dazu den Vorteil haben dynamisch getypt zu sein haben. Doch diese Zuflucht kommt nicht ohne Kosten - Python und Ruby sind in der Ausführung zwei Größenordnungen langsamer [[Citation needed][]]. Clojure Code kompiliert jedoch in JVM Bytecode, ist also mit Java binärkompatibel, hat ähnliche Performanz-Eigenschaften, umgeht aber die ungeliebte Java Syntax.

Andererseits gibt es Programmierer, die die klassische LISP Syntax, die auf S-Expressions fusst[[Citation needed2][]] (wie sie etwa in Common LISP) implementiert ist) für zu Klammer-lastig befinden. Aufgrund dieser speziellen Syntax steht im Jargon FilePaul Graham [b] eine alternative Beschreibung des Akrynyms LISP: "Lots of Irritating Superfluous Parantheses". Auch hier

räumt Clojure auf, indem es ähnlich wie Ruby mit unnötigen Klammern spart. Die Syntax wird um Vectors und Maps

allerdings ohne die Syntax der S-Expressions zu verlassen, sodass am Ende im Vergleich mit anderen Sprachen dennoch überproportional viele Klammern übrig bleiben.

3.3 3.3 Vorteile/Nachteile

Eine der populären Fragen ausserhalb des LISP Umfeldes ist: "Falls LISP wirklich so gut ist, weshalb hat es dann in all den Jahren keine breitere Verbreitung gefunden?"

Der Fragesteller geht hier davon aus ein gutes Argument gegen LISP gefunden zu haben, denn schließlich kann etwas nicht gut sein, nur weil es nicht von der breiten Masse verwendet wird. Dies jedoch ist ein Trugschluss eines Menschen, der im Zweifel keine Grundausbildung in der Logik besitzt. Die Argumentation ist im fachlichen Sinne überhaupt keine, sondern eine empirisch quantative Konfrontation, die mit Qualität nicht zu verwechseln ist.

In den frühen Jahren oder gar Jahrzehnten gab es noch legitime Gründe gegen LISP - damals waren Computer etwa noch nicht ausreichend schnell und mächtig, um dergleichen Hochsprachen auszuführen. In einer Zeit jedoch in der Java seit einem Jahrzehnt breite Verbreitung gefunden hat scheint dieses Argument nicht sehr weitreichend. Heute sind Computer nicht nur schnell genug, um LISPs auszuführen, sie sind sogar im Vergleich mit anderen Sprachen sehr schnell.

Common LISP Kompilate können gar schneller sein als C Kompilate. Das liegt daran, dass es Vorteile hat auf hohem Abstraktionsniveau zu agieren - etwa *Tail Recursion* spart in der Ausführung viel Zeit. Clojure ist nicht ganz so schnell, aber unter der großen Sprachenvielfalt immernoch sehr schnell - es ist etwa 3-5x langsamer als Java. Verglichen mit anderen Hochsprachen wie Ruby oder Python also 2 Größenordnungen schneller.

Heute zählt das Argument der Geschwindigkeit also eher im Vorteil der LISPs - das zweitverbreiteste Argument ist das der Verbreitung, das eingangs des

Kapitesl erwähnt wurde. Dies jedoch ist eine selbsterfüllende Prophezeiung - wenn man nur den Massentrends folgt werden diese rückgekoppelt stärker, wobei popentiell bessere Alternativen ausser Acht gelassen werden. Hier sind wir bei klassischer Monopolbildungstaktiken - nicht bei qualitativen Aussagen über einzelne Entitäten.

Die letzte wichtige Barriere sind Manager im Business, die es sich nicht leisten können Entscheide von Technikern fällen zu lassen - sie bleiben gerne auf der sicheren Seite der sogenannten "Business Best Practices". Auch hier führt dergleichen Entscheidungsfindung zu Monopolbildung. Jedoch bietet Clojure hier einen möglichen Ausstieg - Clojure basiert auf der JVM und ist dort ein Bürger erster Klasse. So können Altprogramme und -bibliotheken ohne Weiteres weiterverwendet werden ohne dabei auf veralteten Paradigmen sitzen zu bleiben. Selbst Microsoft sieht diesen Trend wachsen und liess erst Python auf die .NET Platform, nun entwickeln sie eine eigene funktionale Programmiersprache F#, die unter Entwicklern im Windows Umfeld zusehens stärkere Verbreitung findet.

4 Clojure ist dynamisch

4.1 Dynamisch / Funktional

Die urprüngliche Definition aus den 1950ern von "dynamischer Programmierung" ist nicht mehr deckend mit dem heutigen Verständnis. Damals stand es dafür eine rekursive Funktion zu memoisieren - das bedeutet einen Cache innert einer rekursiven Funktion zu bisherigen Ergebnissen anzulegen. Heute ist es ein schwammig verwendeter Begriff, der gültig ist für eine Untermenge aus folgenden Attributen: * Dynanisches Allozieren von Speicher * Bereits C erfüllt dieses Kriterium mit malloc() und free() * Dynamisch wachsende Datenstrukturen * Java erfüllt dieses Kriterium (etwa mit Vector und ArrayList) * Dynamisches Generieren von Funktionen und Objekten * Auch bekannt als *First Class Functions*. Eine Sprache implementiert First Class Funktionen wenn Funktionen gespeichert, als Argument übergeben und als Ergebnis einer Funktion zurück gegeben werden können. * Dynamisches Aufrufen einer 'eval' Methode * eval ist meist so implementiert, dass Code als String der Methode übergeben wird. Mittels dieses Mechanismus kann die Absenz von First Level Funktionen abstrahieren. * Closures * Eine Closure ist die Möglichkeit innerhalb des Kontextes einer Funktion eine neue Funktion zu schreiben, die Zugang zum ersten Kontext hat. * Macros * Mittels Macros ist es möglich dynamisch zur Laufzeit neuen Code zu generieren. Richtig eingesetzt bieten Macros beispielsweise die Möglichkeit die Syntax der Programmiersprache zu verändern.

Die Definition geht hier also weit auseinander - nach mancher Definition ist schon C eine dynamische Sprache. Meist jedoch wird eine Sprache dynamisch genannt sobald sie *Runtime Types* unterstützt, also Datentypen erst zur Laufzeit bestimmt. Clojure ist unter den LISPs eine der wenigen dynamisch

getypten Sprachen - jedoch unterstützt Clojure *type hints* aus Gründen der Performanz.

Clojure ist eine dynamische Programmiersprache, die auf der Java Virtual Machine und der Microsoft CLR[[Citation needed][]] läuft. Sie ist als universelle Programmiersprache ausgelegt, die die Zügänglichkeit und die interaktive Entwicklung von Scriptsprachen mit Effizienz und Robstusteit paart. Obwohl Clojure in Java Bytecode kompiliert bleibt die Sprache dynamisch, da alle Clojure Funktionen zur Laufzeit verfügbar sind.

Für Clojure bedeutet "dynamisch sein" eine interaktive Umgebung zu bieten. Nahezu jedes Sprachkonstrukt kann vergegenständlicht und somit verändert werden. Darüber hinaus bietet Clojure die klassisch dynamische Entwicklungsumgebug im LISP-Umfeld: ein *REPL*. Ein REPL ist eine Read-Eval-Print-Loop. Das bedeutet, dass Clojure's S-Expressions dynamisch geparst und interpretiert werden. In einem REPL kann wie in Ruby's IRB oder dem Python Interpreter gearbeitet werden. Eien REPL bietet also ein einfaches Konsolen-Interface in das Kommandos eingegeben werden können, um direkt mit den Ergebnissen weiter zu arbeiten.

Clojure ist durch Implementation auf der JVM zwar daran gebunden eine kompilierte Sprache zu sein, doch merkt man als Endanwender (in diesem Fall als Programmierer) nichts, da Clojure neuen Code spontan übersetzt.

4.2 Nicht objektorientiert

Clojure entstand mit dem Hintergrund die Komplexität, die nebenläufige Programmierung mit sich bringt zu verringern. Teil dieser Komplexität ist das Konstrukt der Objektorienterierung. In objektorientierten Sprachen werden Zustände in Objekten gekapselt und über wohldefinierte Schnittstellen veränderbar gemacht. In der Objektorientierung gibt es keine klare Unterscheidung zwischen Zustand und Identität.

Clojure hingegen ergreift eine im ersten Augenblick orthogonal zur Intention eines Programmes liegende Herangehensweise: Daten in Clojures Datentypen sind nicht veränderbar. Dadurch erschlägt Clojure auf einmal die Probleme, die

veränderbare Zustände in Objekten mit sich bringen - beispielsweise Racing Conditions. Darüber hinaus wird das Konzept der Nebenläufigkeit stark vereinfacht - da sich kein Objekt verändert kann es ohne Angst vor Inkonsistenz als Argument verwandt werden.

Imperative Programmiersprachen erlauben die Mutation von Variablen, was für sich genommen eine plausible Herangehensweise ist. Jedoch führt sie bei gleichzeitiger Anwendung zu Konflikten - schon der Begriff "veränderbarer Zustand" ist ein Oxymoron, denn ein Zustand ist ein in diskreter Zeit definiertes Faktum. Ein Zustand kann sich also in diesem Sinne nicht verändern - zu einem anderen Zeitpunkt jedoch kann es völlig neue Zustände geben.

Clojure hingegen fokussiert auf funktionale Programmierung, Nicht-Veränderbarkeit, den Unterschied zwischen Zeit, Zustand und Identität. Objektorientung hat allerdings auch für Clojure interessante Aspekte. So implementiert es einige der Kerngedanken der Objektorientung - allerdings auf ganz eigene Weise: * Polymorphismus * Polymorphismus ist die Fähigkeit einer Funktion je nach Kontext unterschiedliche Fähigkeiten zu besitzen. Hierfür bietet Clojure sogenannte *protocols*. Datentypen können so um Funktionalität erweitert werden. Diese Funktionalität ist ähnlich zu den Mixins aus Ruby. * Subtypen * Obwohl Clojure's Datentypen nicht auf Classen fussen, gibt es eine Möglichkeit Subtypen zu erstellen: ad-hoc Hierarchien. * Kapselung

Einer der Nachteile der strikten Objektorientung ist die enge Kopplung von Daten und Funktion. In manchen Sprachen ist es gar nicht möglich Funktion ohne das elaborate Formulieren von Klassen zu implementieren. Dies bringt eine ganz eigene Form von Komplexität und Weitschweifigkeit mit sich, die von vielen als nicht notwendig angesehen wird.

5 Programmiersprachen in der JVM

5.1 Beispiele

Neben Java bietet die JVM (**Java Virtual Machine**) noch unzähligen weiteren Programmiersprachen als Laufzeitumgebung. Hier nur die prominentesten Vertreter:

Groovy	eine Objektorientierte Skriptsprache.	http:
Scala	eine Sprache mit sowohl Objektorientierten als auch Funktionalen Elementen.	http:
JRuby	Implementierung von Ruby	http:
Jython	Implementierung von Python	http:
Rhino	Implementierung von JavaScript	http:
AspectJ	eine Aspektorientierte Erweiterung von Java	http:
Clojure	funktionale Sprache, LISP-Dialekt	http:

[[http://en.wikipedia.org/wiki/List_of_JVM_languages][]]

5.2 Vorteile/Nachteile

Was aber bewegt Rich Hickey und die Entwickler der anderen Sprachen dazu, ihre Sprachen in die JVM zu integrieren oder zu portieren? Es sind die Enormen Vorteile der JVM, die die wenigen Nachteile bei weitem überwiegen. Dazu zählen unter anderem:

- **Plattformunabhängigkeit:** Die JVM gibt es für sehr viele Plattformen (Linux, Mac, Palm OS, Solaris, Windows, usw.). Das hat zur Folge, dass nur ein Compiler benötigt wird: **Eigene Sprache -> Java Bytecode**.

- **Verbreitung:** Die Verbreitung und Aktualisierung der JVM wird von anderen erledigt. Auch ist der enorme Verbreitungsgrad ein großer Vorteil, da die Benutzer so kein neues System installieren müssen.
- **Sicherheit:** Genau wie bei Java sind die Programme beim Ablauf vom System gekapselt. Das heißt sie laufen in einer Art Sandbox und können so deutlich besser kontrolliert werden.
- **Ressourcenverwaltung:** Da die JVM als Schicht zwischen System und Programm fungiert, sind auch alle Ressourcen gekapselt. Das heißt man muss sich nur begrenzt um Speicherverwaltung kümmern. Eine **Garbage Collection** wird auch gleich mitgeliefert.
- **Bibliotheken:** Es kann ohne großen Aufwand auf die reichhaltigen Java Bibliotheken zurückgegriffen werden, da diese ja meist im **Java Bytecode** vorliegen. So können existierende APIs weiterverwendet werden, aber auch neue Java Frameworks genutzt werden.

Bei all den Vorteilen bringt es aber auch eine paar Nachteile mit sich, eine Sprache in die JVM zu integrieren.

- **Ausführungsgeschwindigkeit:** Durch die zusätzliche Abstraktionsschicht muss man leider Einbusen bei der Performance hin nehmen. Diese halten sich aber dank zahlreicher Optimierungen in Grenzen.
- **Distanz zur Hardware:** Was auf der einen Seite ein Vorteil ist, kann auf der anderen Seite aber wiederum ein Nachteil sein. Durch die zusätzliche Abstraktionsschicht ist es leider nicht möglich, hardwarenah zu programmieren. Das trifft insbesondere auf die Entwicklung von Treibern zu.

6 Probleme und deren Lösungen in unterschiedlichen Programmiersprachen

6.1 Ruby

Listing 6.1: `Thread Race Ruby`

```
1  def inc(n)
2    n+1
3  end
4  sum = 0
5  threads = (1..10).map do
6    Thread.new do
7      10_000.times do
8        sum = inc(sum)
9      end
10   end
11 end
12 threads.each(&:join)
13 p sum
```

Beispielergebnis: 17221

Was ist hier geschehen? Es wurden 10 Threads generiert, die jeweils 10.000x die Variable `sum` inkrementieren. Das Problem hierbei ist, dass bei einem solchen Konzept schnell Dateninkonsistenzen entstehen.

Nehmen wir an Thread #1 inkrementiert die Variable `sum` mittels `inc()`. `sum` steht zur Zeit auf dem Wert 0. `inc(sum)` wird aufgerufen. Nun kommt der Scheduler des Betriebssystemes und gibt Thread #2 CPU Zeit. Dieser fuehrt ebenfalls `inc(sum)` mit dem Wert 0 aus - `sum` wird als 1 gespeichert. Der Scheduler schlägt wieder zu und gibt Thread #1 die Chance seine Berechnung zu beenden. Dieser speichert `sum` nun ebenfalls als 1.

Dieser Effekt wird **Racing Condition** genannt. Er stammt aus dem Englischen und will aussagen, dass mehrere Entitäten in einem Wettlauf um Resourcen stehen. Ohne die Hilfsmittel aus Funktionaler Programmierung müsste die Resource `sum` dagegen abgesichert werden. Dafür werden häufig **Locks** eingesetzt. Bei einem Lock wird ein Codebereich gesichert, indem nur einem Thread dazu Zutritt gewährt wird.

Diese Methodologie funktioniert, jedoch muss jede Stelle an der eine Racing Condition entstehen kann separat gekapselt werden - wird nur eine vergessen sind Bugs im Programm versteckt, die häufig erst spät auftauchen. Darüber hinaus ist es mit Locks möglich sogenannte **Deadlocks** zu erschaffen. Ein Deadlock ist eine Situation in der Thread A auf Ressource R wartet auf die jedoch Thread B Zugriff hat. Thread B jedoch wartet darauf, dass Thread A die Resourcen R mutiert. Diese Situation kann nicht automatisiert gelößt werden - das Programm steht.

6.2 Java

Listing 6.2: `Thread Race Java`

```
1  class MyThread extends Thread {
2    static SharedResource shared;
3
4    public static void main(String[] x) throws InterruptedException {
5      shared = new SharedResource();
6      Thread threads[] = new MyThread[10];
7      for (int i = 0; i <= 9; i++) {
8        threads[i] = new MyThread();
9        threads[i].start();
10     }
11     for (Thread t : threads) {
12       while (t.isAlive()) {
13         sleep(20);
14       }
15       t.join();
16     }
17     System.out.println(shared.sum);
18   }
19
```

```
20   public void run() {
21     for (int i = 1; i <= 10000; i++) {
22       shared.inc();
23     }
24   }
25 }
26
27 class SharedResource {
28   protected int sum = 0;
29
30   public int inc() {
31     return sum++;
32   }
33 }
```

Hier ist das gleiche Problem, nur in Java. Auch hier tritt der Effekt der Racing Condition auf. Der Programmierer muss sich selber darum kümmern, dass `sum` nur von einem Thread gleichzeitig bearbeitet wird. Das kann man erreichen, indem man die methode `inc()` als synchronized deklariert, oder in dem man einen Lock Mechanismus in `SharedResource` implementiert (z.B. über das interface `java.util.concurrent.locks.Lock`).

6.3 Clojure

Listing 6.3: `Thread Race Clojure`

```
1  (def visitors (ref #{}))
2  (defn hello
3    "Hello world function. Remembering previous calls in a transaction"
4    [name]
5    (dosync
6        (if @visitors name
7          (str "Welcome back, " name)
8          (do
9            (alter visitors conj name)
10       (str "Hello, " name)))))
```

In diesem Listing ist der Mechanismus des **Software Transactional Memory** gezeigt. Diesen kennt man aus dem Datenbank-Umfeld. Das Framework

kümmert sich hierbei darum, dass die Resource `visitors` geschützt wird. Sie kann nicht ohne einen **dosync** Block mutiert werden. Dieser Block wiederum stellt wiederum sicher, dass Mutationen transaktional ablaufen.

7 Zusammenfassung

7.1 Alleinstellungsmerkmale

Ein wirklich Alleinstellungsmerkmal im klassischen Sinne ist bei Clojure schwer zu finden. Es ist vielmehr die Kombination von schon vorhandenen Elementen, die es so noch nicht gibt. Es gibt andere funktionale und dynamische Sprachen, LISP-Dialekte sowie Sprachen die sich sehr gut zur Lösung von Nebenläufigkeitsproblemen eignen. Und das alles gibt es sogar auf der JVM. Aber es gab bisher keine Sprache, die das alles Vereint.

7.2 Einsatzgebiete

Prinzipiell lässt sich jedes Problem mit jeder Programmiersprache lösen, solange diese **Turing complete** ist. Es gibt aber sehr große Unterschiede, was den Aufwand betrifft, den man dafür betreiben muss. Und der Aufwand beinhaltet nicht nur das schreiben des Codes selbst, sondern auch die Installation einer Laufzeitumgebung oder das Compilieren für unterschiedliche Plattformen.

Durch die Bindung an die JVM macht der Einsatz von Clojure schon einmal insbesondere da Sinn, wo schon mit einer Java Umgebung gearbeitet wird. So erspart man sich die Wartung einer separaten Laufzeitumgebung. Warum dann nicht gleich Java? In Bereichen wo viel mit Nebenläufigkeit gearbeitet wird, kann man sehr einfach auf Locks, sychronized-Blöcke, Race Conditions, explizit als volatil zu deklarierende Instanzvariablen oder ConcurrentModificationExceptions verzichten, wenn man auf Clojure setzt. Der allgemeine Entwicklungsablauf ist in Clojure durch die REPL auch deutlich dynamischer und interaktiver als in Java.

7 Zusammenfassung

Clojure eignet sich auch hervorragend zur Lösung Mathematischer Probleme, da das funktionale Konzept erheblich näher an die Mathematik angelehnt ist, als imperative Konzepte. Funktionen in Closure entsprechen auch eher einer mathematischen Funktion als z.B. Funktionen in C.

Quellenverzeichnis

of Computer Science Nottingham

http://www.cs.nott.ac.uk/~gmh/faq.html 2.1

Paul Graham a

http://www.paulgraham.com/mcilroy.html 3.1

Paul Graham b

http://www.paulgraham.com/jargon96.html 3.2

wikipedia.org

http://de.wikipedia.org/wiki/Funktionale_Programmiersprache 2.1