Clojure

Brandon Mohajeri
brandon.mohajeri@gmail.com

Abstract: This intent of this paper is to go into a deep analysis of the Clojure programming language. Beginning with a brief history and overview, we will look into some of the key features of the Clojure Programming language, including concurrency, JVM interoperability, the REPL, and Functional programming. There will be a review of the project program, which will demonstrate aspects of Clojure as well as areas of improvement in the code. Following will be a brief conclusion summarizing the importance of Clojure and my personal experience.

1 Introduction

With no experience in Clojure, I wanted to create a program which could be completed and enhanced in the allotted time. I wanted to introduce myself to a functional programming language because I have never worked in that type of environment before. The motivation behind the project was to build a practical application that can be used every day. The intent was to create an IMDB movie manager, which is reviewed in the programming experience section of this paper.

2 Project Language Analysis

Clojure is a modern day variation of the Lisp programming Language with an emphasis on functional programming. Created by Rick Hickey in 2007 as a general-purpose programming language using traditional object oriented techniques, he wanted Clojure to be symbiotic with the Java platform as well as be designed for concurrency. The community surrounding Clojure is very active due to the fact that the development process is community-driven. A majority of programmers use Leinengen for project automation and support. While Clojure is still a young language, it bases it's best practices with its roots in 50 years of LSIP and 15 years in Java history.

2.1 Concurrent Programming

Built on a focus on concurrent programming, it is no surprise that Clojure has multiple support systems for this parallel type programming. With parallel programming comes many benefits including the ability to simultaneous access multiple applications and the access to immutable persistent data structures.

Clojure encompasses a completely different set of concurrent programming models Let us review the following four.

2.1.1 Thread Local Variables

The most basic type of Clojure concurrency model is a simple declaration of variable. The benefits of this is that we are able to prevent read and write access from another thread. View example Below.

```
(def message “Hello World”)
```

2.1.2 Atoms

An atom is a completely synchronous variable in Clojure whose state is able to be changed because it is in the main thread. For example, if we were to have a function where we change the value of an atom, all threads will see the new value upon return. View Example Below.

```
(def myNum (atom 5))
(swap! a 10)
;swap! is used to swap the value of atom to bed
```

2.1.3 Transactional References

Clojure uses concurrency control in order to ensure that correct results for concurrent operations are generated quickly. It is through a software transactional memory(STM), where transactional references are ensured safe shared use of mutable storage locations. All actions on References are atomic, consistent, and isolated, analogous to database transactions. The benefit of STM is that each transaction can be viewed in isolation as a single-thread, which greatly simplifies multithreaded programs. The main constructs for STM are ‘ref’ and ‘dosync’. Example Below

```
(def savings (ref 100))
(def checkings (ref 200))

(def transfer-money [savings checking 300]
       (dosync
              (alter savings - amount)
              (alter checking + amount)
              amount))




(transfer-money savings checking 20)    ; transfer 20
(println @savings @checkings)           ;prints 80, 220
```

Refer to Reference 7

2.1.4 Agents

Agents provide shared access to mutable state. Agents provide independent, asynchronous change of individual locations and are bound to a single storage location. These agents start producing background threads which prevent the JVM to shutdown, so it is important to terminate these threads. View example below.

```
(def counter (agent 0))  ;Agent created
(send counter inc)       ;Send request to change value
(print counter)          ;Prints 1
```

While there are many upsides to concurrent programming in Clojure, there are very few: 1) Threads can become expensive, 2) Difficulty Debugging 3) Depending on your application, concurrency can actually slow down your program.

2.2 REPL

Clojure makes use of a read-eval-print loop (REPL). This is able to take in a single user input and evaluate it. This makes the learning curve for Clojure much smaller, allowing users to test and maintain immediate results of bits of code. The REPL is popular in many Lisp languages.

2.3 Functional Programming

Clojure is a functional programming language, providing functions as first-class objects which are impure. Clojure was built on the fact that most parts of most programs should be functional.

With a functional programming language like Clojure, we are greatly able to reduce our lines of code. This difference became clear to me when I compared my experience with Java and Clojure. This makes dealing and maintaining code much simpler.

First Class Functions: Clojure uses its keyword ‘fn’ to declare functions.

```
;Function which takes a parameter and rounds it
(def round
    [d precision]
    (let [factor (Math/pow 10 precision)]
      (/ (Math/floor (* d factor)) factor)))
```

Refer to Reference 5

Multi-arity: Functions: Functions in Clojure are able to have a set of arguments.

```
(def tax-amount
    ([amount]
          (tax-amount amount 25))
   ([amount rate]
          (Math/round (double (* amount (/ rate 00))))))
```

2.4 JVM Interoperability

Unlike many new programming languages Clojure has access to stable libraries, allowing the programmer to focus on practicality.

A key feature, and another focus during its creation was interoperability with Java and ability to run on the JVM. Running on the JVM provides many benefits such as portability, stability, performance, as well as access to existing Java libraries. This allows Clojure to support file I/0 and many more. Not only can Java code be called by Clojure, but Clojure code can also be called by Java. While there are many benefits for Clojure to be hosted on the JVM, there are few negative aspects as well. A major one being is that it slows down your program.

3 Programming Experiments

I chose this program for my project simply because I love movies, but I didn't have one hub to manage and access the titles the way I wanted. With little experience in web scraping and parsing large amounts of data, I wanted to tackle a movie manager program which would have the ability to scrape IMDB for movie statistics (i.e rating, genre, reviews, ect.), display them to the user, and allow the user to save the movie to a file to display later.

3.1 Slurp HTTP Get

Clojure's 'slurp' function is the simplest way to read in a file, and is a result of the clojure.java.io/reader library. While slurp is simple and quick to use, it is not recommended for larger files because it reads the file into memory. The result is also returned as a single string with no line breaks or any other delimiter.

```
(defn findMovie [movieName param]
          (def movie (slurp ('link to film url')))
```

Here we use the 'defn' keyword, which evaluates the argument every time its called, unlike 'def' in order to create a function which takes two parameters and calls the

slurp method which scrapes the IMDB page for the HTML source, and assigns it to movie.

3.2 Parsing the Data

One of the bigger issues I discovered in this Clojure program was parsing the data. While this was an issue on my early on in my development process, once I was able to familiarize myself with how Clojure's, split, replace, sub, and nth namespaces, it became very convenient and simple. View code samples below.

```
(def movieNames (clojure.string/replace (subs (nth x 0) 10 (
.length (nth x 0))) "\"" ""))

(def movieRating (clojure.string/replace (subs (nth x 2) 8
(.length (nth x 2))) "\"" ""))
```

3.3 I/O

I was able to utilize Java's libraries by including

```
(use 'clojure.java.io)
```

In the head of my Clojure program. This added a great deal of familiarity and comfort seeing as how I have used the BufferedReader before, and reading input was made simple. The same applies for the writer. My program takes the user input multiple times in the console to determine which function to call upon next. When the user wants to save a movie, the program calls the java IO writer library and appends the results to a local file for the user to call upon later. The Java IO library is also called upon when the user wants to view a list of movies.

Reader:
```
(let [reader (java.io.BufferedReader. *in*)
      ln (.readLine reader)])
```

This section of the Program uses the Java library to take in the user input. This was made very simple and easy with the Java IO libraries.

Writer:
```
(if (= (read-string param) 1)
    (with-open[w(clojure.java.io/writer
    "w.txt" :append true)]
    (.write w (str " DATA " ))
```

Here Clojure conducts and if loop, and like any LISP language is evaluating the expression given in a lexical context. This is great for logic programming. If the expression returns true, Clojure calls upon the java library and appends the film data to the file.

3.4 Environment

I used Sublime Text for my text editing environment then used my terminal to run the popular Clojure task runner, Leiningen. Leiningen is an important aspect in the Clojure development process, with the ability to handle 1) Project compilation 2) Dependency management 3) Running tasks 4) Deployment. This script can simply be run on your terminal. Overall, my experience with Leiningen was positive and made deploying code much simpler. There is also quite a large community around it providing support through forums.

3.5 Code Origionality

With an original idea, I was able to produce a fully unique code from scratch. Due to the strong Clojure community, I was able to optimize and take different approaches to my program with online resources and forums.

4 Conclusions

After spending the last four months with Clojure, I have learned about Lisp Programming and concurrent programming. More specifically, the power of parallel programming and how much it can optimize your program speed.

While there were many benefits in programming with Clojure, but I did find a few aspects of it to be difficult. I would have liked to add a GUI to my program, but there were not many GUI support systems for Clojure. The best I came across was Seesaw, which is a Clojure library that is a GUI wrapper for Clojure's Swing, making the code much easier to deal with. Along with the lack of a simple GUI support, Clojure's stack trace was surprisingly difficult to follow with its verbose errors. This made debugging difficult and time consuming. Other than the GUI, stack-trace, and difficulty to parse data, there were no real issues limiting me from creating my program.

I achieved a lot through this project. Not only did I tackle, what I believed to be a difficult language in the first place, I was also able to go beyond my original project outline and make a fully useable and useful Movie Finder/Manager console application. If revisions were to be made in my program, I would utilize Clojure's use of Concurrency. I want to implement multithreads in Clojure in order to send an HTTP request and parse the data at the same time so that there is less delay when printing to the console.

Clojure is not a great with first impressions, but once you gain a grasp of the syntax as well as operating in a functional programming environment it becomes a brilliant programming language. It is not until you fully gain a grasp of concurrency that you can fully utilize the power of the language.

References

1. "The Clojure Philosophy." *Dr. Dobb's*. N.p., n.d. Web. 14 Dec. 2015.
2. "Clojure and Concurrency." *Clojure and Concurrency*. N.p., n.d. Web. 14 Dec. 2015.
3. "Clojure." - *Home*. N.p., n.d. Web. 14 Dec. 2015.
4. Chas Emerick - Brian Carper - Christophe Gran - O'Reilly - 2012
5. http://clojure-doc.org/articles/language/functions.html
6. http://java.ociweb.com/mark/clojure/article.html#Intro
7. http://sw1nn.com/blog/2012/04/11/clojure-stm-what-why-how/