Steps towards teaching the Clojure programming language in an introductory CS class (extended abstract).

Elena Machkasova
University of Minnesota, Morris
elenam@morris.umn.edu

Stephen J Adams
adams601@morris.umn.edu

Joe Einertson
University of Minnesota, Morris
eine0017@morris.umn.edu

The Clojure programming language is a new language in the LISP family that is gaining rapid popularity in industry due to its elegant design and suitability for concurrent programming. We argue that Clojure has benefits for being taught as a programming language in an introductory course. We discuss several challenges that need to be overcome in order to teach Clojure to students with no programming experience, such as modifying error messages and providing a beginner-friendly development environment, and describe our work in progress towards these goals. We also discuss conceptual approaches towards teaching Clojure to beginners that make use of Clojure sequence abstraction.

1 Introduction

The Clojure programming language is a new language in the LISP family that is gaining rapid popularity in industry due to its elegant design and suitability for concurrent programming. Clojure has been incorporated into upper-division undergraduate courses, including those at University of Minnesota, Morris (UMM). In this paper we argue that Clojure has benefits for being taught as a programming language in an introductory course (section 3). We discuss several challenges that need to be overcome in order to teach Clojure to students with no programming experience, such as modifying error messages and providing a beginner-friendly development environment. We describe our efforts towards these goals (section 4). We also discuss conceptual approaches towards teaching Clojure to beginners that make use of Clojure sequence abstraction (section 5).

The project on adopting Clojure as a language for an introductory class is work in progress conducted at UMM as a combined effort of faculty, students, and alumni. Currently we are implementing and improving beginner-friendly Clojure development environment and utilizing testing feedback from two sophomores (computer science majors) who have no prior experience with Clojure, and one of whom has no experience with functional programming. The introductory level class that uses Clojure is scheduled in the Fall semester of 2013.

2 Overview and history of Clojure

Clojure is a dialect of Lisp developed by Rich Hickey and released in 2007 [**?**]. Clojure was developed to answer the need for a practical functional language that supports convenient, efficient, and safe programming for concurrency. Clojure provides a rich set of immutable data structures, augmented with several kinds of mutable reference types. Clojure in its classical implementation is complied into Java bytecode and can be fully integrated with Java, both by using Java objects and methods and by providing code that can be invoked from Java. It also provides a REPL (Read Eval Print Loop) for interactive development.

Submitted to:
TFPIE 2013

Clojure was carefully designed with efficiency in mind, and provides constructs for tail call optimization, data structures with fast traversal to any element that maximize sharing when a modified version is created, and efficient handling of lazy structures, among other mechanisms for providing efficiency. As a result, Clojure is as fast as Java on their shared underlying platform (the JVM) . It is also much more effective for concurrent execution due to immutability of data structures and a significantly reduced need for locking and synchronization.

Because of its convenience, efficiency, and elegant design Clojure is rapidly gaining popularity in the software development community. This past year Clojure was rated as "Adopt" by the ThoughtWorks Technology Radar; there were four major conferences dedicated to Clojure, two in the US and two in Europe, and the blog aggregator "Planet Clojure" includes over 400 blogs [**?**].

3 Benefits of teaching Clojure as the first language

In this work we explore incorporating Clojure into an introductory undergraduate computer science course. We describe the course setup at UMM which is fairly typical for a small university. At UMM we have about 3.5 contact hours per week that include lecture and lab time and other activities, e.g. group discussions. There are 30-40 students in the course, including CS majors/minors, undecided students with interest in CS, and those taking it to satisfy an elective or a general education course. The majority of the students (even those who intend to major in CS) have no prior programming experience and very little understanding of the process of computing.

The course focuses on problem solving in general and its applications to computer science. Students work on understanding a problem's requirements and gradually develop solutions in a programming language, with an emphasis on effective design and testing strategies. They learn about language syntax and behavior, key programming concepts such as conditionals and functions, data representation in a program, and similar introductory-level concepts. Mastering the concept of recursion is one of the key learning objectives in the course since it is a basis of many important data structures and algorithms. Towards the end of the course students develop their own implementations of simple searching and sorting algorithms and are informally introduced to the notion of big-O. Course exercises include both concrete assignments and open-ended projects, such as graphics and simple game development. Students often work in groups, which promotes inter-personal communication and a code style that conveys intention.

The benefits of using a functional programming language in introductory computer science classes are well-known [**?**]. Functional languages focus on functions as programming units, provide abstraction, generalization, and modularity, and give a better understanding of recursion, one of the key learning goals in an introductory course. Functional languages tend to have simpler, more uniform syntax that students master quickly. There are several successful introductory functional languages curricula, such as the How To Design Programs course with Racket [**?**] and introductory courses with Haskell, e.g. [**?**].

Developing a similar undergraduate curriculum based on Clojure would provide additional benefits. Clojure combines a rich set of features with an elegant uniform underlying model based on abstraction. Clojure "collections" (i.e. data structures) include lists, vectors, sets, and hashmaps. However, all of these collections can be worked with as sequences (see section 5). Abstraction-based language model teaches students to use and appreciate abstraction and modularity and makes it easier for students to learn language features and libraries not covered in class on their own.

Clojure is also beneficial in preparing students for moving forward in the CS program. Clojure is fully integrated with Java since both compile to Java bytecode: Java code can be called from Clojure and vice versa. Integration with Java does not have an efficiency penalty for either of the languages.

Original type	Beginner-friendly type
java.lang.Number	a number
java.lang.Integer	a number
java.lang.Double	a number
java.lang.String	a string
java.lang.Character	a symbol
clojure.lang.Symbol	a symbol
clojure.lang.ISeq	a sequence

Table 1: Sample type conversion

Students can integrate Clojure into projects developed in Java or labs that use pre-written Java code in their upper-division classes, allowing them to incorporate functional style of programming as needed, even in a part of a project.

A related benefit is Clojure's suitability for concurrency. Clojure provides mechanisms for multi-threading that do not require explicit thread synchronization, locks, or adjustments to the number of processors. While Clojure concurrency cannot be covered in an introductory course, background in Clojure makes it easy for students to learn this material in a later course or independently.The students would already be familiar with key concepts, such as immutable persistent data structures. Clojure fits the recent emphasis on teaching concurrency in undergraduate curriculum (e.g. [**?**]) perfectly.

Clojure has a friendly, well-developed community with online users-maintained documentation and examples, multiple blogs, several Google groups and an IRC channel, numerous open-source projects, and several excellent quality books (although not at a beginning programmer level). All these resources would enable students to continue their Clojure development past the introductory course.

4 Technical challenges of teaching Clojure as the first language

The key technical (as opposed to conceptual) challenges of teaching Clojure to beginners stem from the fact that it was not designed as a language for beginners. Clojure currently does not have a beginner-friendly development environment. Another significant problem is that Clojure error messages come directly from Java and mention Java types and other notions that do not appear directly in Clojure (Clojure is implicitly typed) and are not understandable to beginners. Below we describe our work developing a custom environment that addresses these problems.

Clojure error messages are Java exceptions and therefore mention Java types (some native to Java and some coming from Clojure's implementation in Java) that are unclear to beginners, especially because Clojure is dynamically typed with no need need to specify types explicitly. Types used in Clojure implementation form a rich Java hierarchy; for instance, numerous types implement an interface that represents a Clojure sequence type. These details do not matter to students, but would get in the way of their learning experience. There are other cases when an error message does not make sense to new students at all, such as `NullPointerException`.

4.1 Error messages

The run-time environment that we are developing intercepts Clojure errors by providing a `try/catch` block around students' code and reformatting error messages. As an example, consider the following

Exception type	Original message	Beginner-friendly message
`ClassCastException`	X cannot be cast to Y	Attempted to use X', but Y' was expected.
`IllegalArgumentException`	Don't know how to create X from: Y	Don't know how to create X' from Y'
`IndexOutOfBoundsException`	n	An index in a sequence is out of bounds. The index is: n
`NullPointerException`	–	An attempt to access a non-existing object (NullPointerException)

Table 2: Sample message wording conversion. X, Y are the original types, X', Y' are the corresponding beginner-friendly types.

error message for `(cons 2 3)`, i.e. at an attempt to conjoin an element onto something that is not a sequence and cannot be converted to one:

```
IllegalArgumentException Don't know how to create ISeq from:  java.lang.Long
```

While the wording of the message is reasonably clear, the types used in it would not be understandable to beginner programmers at all. `ISeq` is an interface that represents a Clojure sequence, but beginner students are not familiar with interfaces. Numbers in Clojure can be represented as a variety of types, and it is quite common for them to be represented as the `java.lang.Long` type. The exception type itself also is not useful to beginners for the most part. The error can be made clearer for beginners if we replace type names by those that make sense to beginners and remove the exception type:

```
Error:  Don't know how to create a sequence from a number.
```

Table 1 shows more examples of type names conversion. If no type match can be found, either by a lookup or by checking interfaces that a type implements, then we would display the type as an "unrecognized type" followed by the actual type. However, as we expand our type dictionary and continue testing, these cases occur less and less frequently. There are other error messages that require more beginner-friendly wording; see table 2 for examples.

Catching and reformatting Clojure error messages also allows us to simplify the standard stack trace. A complete stack trace of an error message often contains over a hundred Clojure and Java function calls. The way we approach this issue is by filtering out everything in the stack that is not a part of the student's project (such a project may include some of our own functions added for simplicity). This allows students to see the part of the stack trace that directly relates to their code, and nothing else. The error message filtering has worked well in tests and gives messages that make sense to beginners.

4.2 Development environment

Even though new students will start off by simply writing code in a basic text editor, their code will typically be a part of a project. We will provide a template project that contains all of the necessary libraries and project settings and a file for the students to write their code. It includes the exception-catching and reformatting and predefined Clojure functions that we need to supply for students to use in their own code. We also include libraries, such as a turtle graphics library [**?**] and a Clojure wrapper for Java Swing (a Java GUI library), called seesaw [**?**]. The environment makes it possible to run students' code as a complete program (designated in a project setup) as well as in a REPL for interactive testing.

Our project, like many Clojure projects, is managed via a command-line tool called Leiningen [**?**].

Leiningen handles dependencies, controls compilation, and allows one to run either a function of the project designated as "main" or start a REPL with all of the project's code loaded.

A good Clojure development environment for beginners must have the following characteristics:

- Provide at least basic syntax highlighting and formatting.
- Provide at least the basics of Clojure project management.
- Report errors in a reasonable way, with line numbers.
- Be convenient and easy to understand.
- Be reliable and bug-free.

Unfortunately, there is not a perfect IDE for Clojure. Advanced text editors, e.g. `emacs`, and the plugin for the common Java IDE, Eclipse, are too complicated for beginners. Several text editors intended specifically for Clojure are still in development stages and are unreliable or lack functionality. We decided to use the text editor `jEdit` [**?**] which has a plugin for Clojure syntax highlighting, and then manage projects with Leiningen's terminal commands. We are working on creating a jEdit plugin that allows to call Leiningen commands. We also work to enabe beginner-friendly error reporting not only when running `main`, but also for compilation errors and those generated while using the REPL. Currently we get original error messages in these cases since they are outside of the try/catch block in `main`.

4.3 Other language modifications

There are a few functions that may be confusing for beginners. For instance, the `contains?` function returns `true` if a key is present in a data structure, and not the element. It would return `true` if passed any vector of with a length of at least 2 and the number 1, regardless of the actual elements of the vector, since indices in a vector are considered keys. Providing a function with a more intuitive name, such as `contains-value?`, would solve this issue.

5 Approaches to teaching Clojure to beginners

Clojure provides a rich set of immutable collections. Each collection's implementation is optimized for efficiency based on the intended use. For instance, lists are singly-linked lists, similar to Common Lisp lists. Creating a new list from an existing one can be done in a constant time when an element is added or removed from the beginning by taking advantage of sharing the rest of the elements. However, adding an element at the end takes linear time since the entire list needs to be copied. Vectors are a highly efficient data structure for both insertion and deletion at any position in a logarithmic time. They are implemented as shallow trees so that everything that is not on the path to a changed element can be shared between the old vector and the new one. A function `conj` that adds an element to a collection, returning the result as a collection of the same type as the original, positions the element in the most efficient way for this type of a collection. Thus it adds it at the beginning for a list and at the end for a vector. While this makes sense from the language implementation standpoint, this behavior is very confusing for new programmers.

Most functions on collections, however, are not collection-specific: they take any collection that can be converted into a sequence of elements in some order, and return a sequence, rather than a specific collection. For instance, `map` that maps a function over all elements of a collection, returns a sequence regardless of what kind of collection was passed to it. Sequences are stored as list-like structures, or as a different implementation (e.g. as lazy sequences), but they all function exactly the same in relation to all function that they can be passed to.

Encouraging students to program in a collection-independent way, i.e. using a sequence abstraction, helps them focus on the concepts, rather than specifics of collections implementations. We provide collection-independent functions in a few cases when they do not exist, such as `add-last` that adds an element to a collection at the end, returning the result as a sequence. We also provide examples of handling data in a collection-independent way since most examples in Clojure forums and documentation use collection-specific functions for efficiency. Students will see both collection-specific and collection-independent functions which would provide a good understanding of abstraction and its benefits.

6 Conclusions

Clojure is a promising candidate for an introductory CS course. Its rich collection of data structures and focus on abstraction teaches students good programming skills. The growing use of Clojure in industry means that there is an active and helpful community surrounding the language which helps students to continue Clojure development after the introductory course. However, in order to adopt Clojure as a language for an introductory class one needs to tackle a few challenges, such as confusing errors messages and a lack of beginner-friendly development environments.

7 Acknowledgments

The authors thank Jon Anthony, Brian Goslinga, and Simon Hawkin for helpful discussion and suggestions and Max Magnuson and Paul Schliep for thorough testing of our development environment.